LEABHARLANN CHOLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH | TRINITY COLLEGE LIBRARY DUBLIN
Ollscoil Átha Cliath | The University of Dublin

Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Hardware Support for Power and Area Efficient Construction of High-Quality Bounding Volume Hierarchies

A Dissertation

Submitted to the office of Graduate Studies

of

The University of Dublin

Trinity College

in fulfillment of the requirements

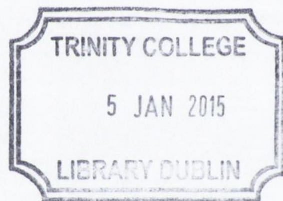
for the Degree of

Doctor of Philosophy

by

Michael J. Doyle, B.A. (Mod.)

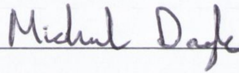
August 2014


TRINITY COLLEGE
5 JAN 2015
LIBRARY DUBLIN

Thesis 10745

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other University. Furthermore this thesis is entirely my own work and I agree that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

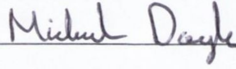


Michael J. Doyle

5th August 2014

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.



Michael J. Doyle

5th August 2014

Hardware Support for Power and Area Efficient Construction of High-Quality Bounding Volume Hierarchies

Abstract

Rendering methods based on ray-tracing hold the promise of great realism for interactive applications. However, these algorithms still involve a considerable computational cost relative to the capabilities of today's hardware. In response to this, a large body of research directed towards improving the efficiency and performance of interactive ray-tracing systems has emerged in recent years. Although modern graphics processors have evolved to support ray-tracing through general-purpose compute capabilities, this approach requires engaging a large amount of hardware resources for optimal performance. Moreover, recent developments, such as the rise of mobile computing, and the emergence of power-efficiency as the primary limit to processor scalability, have placed power near the top of the agenda.

One approach to improving the efficiency of ray-tracing which has received comparatively little attention is the design of specialised ray-tracing hardware. The research that does exist on this topic has consistently demonstrated that significant performance and efficiency gains can be achieved with dedicated microarchitectures. However, previous work on hardware ray-tracing has focused almost entirely on the traversal and intersection aspects of the pipeline. As a result, the critical aspect of the management and construction of acceleration data-structures remains largely absent from the hardware literature.

This work proposes that acceleration data structure construction is amenable to hardware specialisation. To demonstrate this, this thesis presents the first published microarchitecture for the construction of high-quality bounding volume hierarchies for ray-tracing. Cycle-accurate simulations show that the design achieves significant improvements in raw performance compared to state-of-the-art software implementations. Furthermore, these results are achieved using minimal hardware resources, representing a large efficiency improvement compared to existing methods. Power estimates of the architecture reveal the potential for the design to improve power-efficiency, which may help future graphics processor designs to reduce predicted technology-imposed utilisation limits.

Acknowledgments

Firstly, I would like to thank my supervisor, Dr. Michael Manzke. I have worked with Michael for many years, and have found his advice to have contributed greatly to my skills as a researcher. He opened up many opportunities for me over the years, all of which have been critical to my academic development.

Dr. Ross Brennan has been an absolutely invaluable source of knowledge and advice, both in terms of his expert understanding of hardware design, as well as his experience with the PhD process. I do not recall a single occasion where he was not willing to go out of his way to help me.

Dr. Muiris Woulfe has always been there to help. His excellent appreciation of both hardware and software issues made him uniquely qualified to comment on my work. In addition, his uncanny proof-reading ability greatly contributed to the accuracy and integrity of the final manuscript.

Dr. David Gregg has been of great assistance over the years. David has a great ability to understand and give valuable input on other people's work, even on topics outside of his own field.

I cannot forget Colin Fowler, who I have worked alongside for five years. His deep understanding of ray-tracing and his expert-level software skills have been an invaluable source of assistance over the years. Colin and I have spent innumerable hours, sometimes into the early hours of the morning, subsisting on takeaway food and discussing ray-tracing on the whiteboards of Lab 0.03.

I would like to thank Stephen Kenny for being the only person with the know-how to deal with the Mentor Graphics software license!

I would like to thank my examiners, Dr. John Dingliana and Dr. Tomas Akenine-Möller, for offering their time and expertise to evaluate my work. Both are experts in their respective fields, and it was an honour to present and discuss my work in depth with them.

Finally, I would like to thank my family, who have supported and nurtured my interests over the years. Looking back on my childhood, I think of the many sacrifices they made and acts of generosity they showed that have contributed directly to this moment.

MICHAEL J. DOYLE

The University of Dublin
Trinity College
August 2014

Contents

Abstract vii

Acknowledgments ix

Table of Contents xi

List of Tables xiii

List of Figures xv

List of Acronyms xix

1 Introduction 1

1.1 The Importance of Spatial Index Structures 1
1.1.1 Ray-based Global Illumination 2
1.1.2 Rasterisation 3
1.1.3 Collision Detection 3
1.2 A Case for Hardware Support 4
1.3 Research Question 6
1.4 Contribution 6
1.5 Publications 6
1.5.1 Directly Relevant 6
1.5.2 Indirectly Relevant 6
1.6 Summary of Chapters 7

2 Background 9

2.1 Rendering Algorithms 9
2.2 Ray-based Global Illumination 10
2.2.1 Ray-Casting 10
2.2.2 Whitted Ray-Tracing 11
2.2.3 Path-Tracing 12
2.2.4 Photon Mapping 14
2.3 Spatial Index Structures 15
2.3.1 Flat vs. Hierarchical Structures 15
2.3.2 Spatial Subdivision vs. Object Partitioning 17
2.4 Bounding Volume Hierarchies 18

2.4.1 BVH Construction . . . 18
2.5 Special Purpose Ray-Tracing Hardware . . . 29
2.5.1 Fixed-Function Designs . . . 29
2.5.2 Fully Programmable Designs . . . 31
2.5.3 Hybrid Fixed-Function/Programmable Designs . . . 33
2.5.4 Commercial Products . . . 35
2.5.5 Volume Rendering Architectures . . . 36

3 Design and Implementation **39**
3.1 Broad Design Decisions . . . 39
3.2 System Overview . . . 41
3.3 Subtree Builder Microarchitecture . . . 43
3.3.1 Sequence of Operations . . . 45
3.3.2 Partitioning Unit Datapath . . . 48
3.3.3 Binning Unit . . . 50
3.3.4 SAH Calculator . . . 53
3.3.5 Tree Output and Format . . . 58
3.4 System Integration and the Upper Builder . . . 62
3.4.1 Upper Builder . . . 63
3.5 Implementation . . . 65
3.5.1 Primitive Modules . . . 65
3.5.2 Higher-Level Modules . . . 65
3.5.3 DRAM Model . . . 66
3.5.4 Software and Testing . . . 66

4 Evaluation **69**
4.1 Test Scenes . . . 69
4.2 Subtree Builder Analysis . . . 69
4.2.1 Performance Analysis . . . 71
4.3 Full BVH Builder Analysis . . . 75
4.3.1 Performance Analysis . . . 76
4.3.2 Memory Behaviour . . . 77
4.3.3 Tree Quality . . . 79
4.3.4 Area and Power . . . 82

5 Conclusion and Future Work **89**
5.1 Conclusion . . . 89
5.2 Future Work . . . 89

A Additional Circuits **91**
A.1 The Ray-Tracing Pipeline . . . 91
A.1.1 AABB Intersection Unit . . . 92
A.1.2 Triangle Intersection Unit . . . 94

Bibliography **97**

List of Tables

3.1 Floating Point IP Characteristics 65
3.2 Memory IP Characteristics 65
3.3 High-Level Unit Characteristics 65
3.4 mike_rt command-line options 67

4.1 Tested HWBVH Instantiations. 75
4.2 Memory Footprint 78
4.3 BVH Quality Measures 80
4.4 SAH Costs and Node statistics 81
4.5 Subtree Builder Hardware Cost 83
4.6 HWBVH-C Upper Builder Hardware Cost 83
4.7 HWBVH-HP Upper Builder Hardware Cost 83
4.8 System totals for HWBVH-C and HWBVH-HP. 83
4.9 InCyte Chip Estimator parameters. 84
4.10 InCyte Chip Estimator area and power estimates. 84
4.11 Total energy expended during hierarchy construction. 85
4.12 Total power consumption during hierarchy construction. 85

List of Figures

1.1 Ray-Tracing and Collision Detection . . . 2

2.1 Ray Casting . . . 11
2.2 Ray based Global Illumination . . . 12
2.3 Ray-traced Rendering . . . 13
2.4 A simple bounding volume . . . 15
2.5 The Uniform Grid . . . 16
2.6 The kd-tree . . . 16
2.7 The Bounding Volume Hierarchy . . . 19
2.8 Binned SAH . . . 22
2.9 Morton Code Ordering . . . 24

3.1 Overview of the BVH Builder Microarchitecture . . . 41
3.2 Overview of the Subtree Builder Microarchitecture . . . 43
3.3 Partitioning Unit Flow Chart . . . 46
3.4 Organisation of threads in the Subtree Builder . . . 48
3.5 Partitioning Unit Datapath . . . 49
3.6 Binning Unit Group . . . 51
3.7 Binning Unit . . . 52
3.8 SAH Calculator . . . 54
3.9 SAH Uni-collector . . . 56
3.10 SAH Uni-collector Data Layout . . . 56
3.11 SAH Cost Evaluator . . . 57
3.12 BVH Index Selection Logic . . . 60
3.13 System Integration . . . 62

4.1 Test Scenes . . . 70
4.2 Design Space Exploration of the Subtree Builder . . . 72
4.3 Effect of Leaf Size on Construction Performance . . . 73
4.4 Construction Performance . . . 77
4.5 Memory Traffic Breakdown . . . 78

A.1 The Ray-Tracing Pipeline. . . . 92
A.2 Microarchitecture of the AABB intersection Unit . . . 93
A.3 Möller-Trumbore Ray Triangle Intersection Unit . . . 94

A.4 Valid Check for Möller-Trumbore Intersection Unit 95

List of Algorithms

1 Generalised pseudocode for top-down recursive BVH construction. 20
2 Pseudo-code for the naive agglomerative clustering algorithm. 25

List of Acronyms

AABB	axis-aligned bounding box.
ASIC	application-specific integrated circuit.
BIH	bounding interval hierarchy.
BVH	bounding volume hierarchy.
FPGA	field-programmable gate array.
HLBVH	hierarchical linear bounding volume hierarchy.
IP	intellectual property.
ISA	instruction set architecture.
LBVH	linear bounding volume hierarchy.
OBB	oriented bounding box.
RAPL	running average power limit.
RTL	register transfer level.
SAH	surface area heuristic.
TDP	thermal design power.
VHDL	VHSIC Hardware Description Language.
VLIW	very long instruction word.

Chapter 1

Introduction

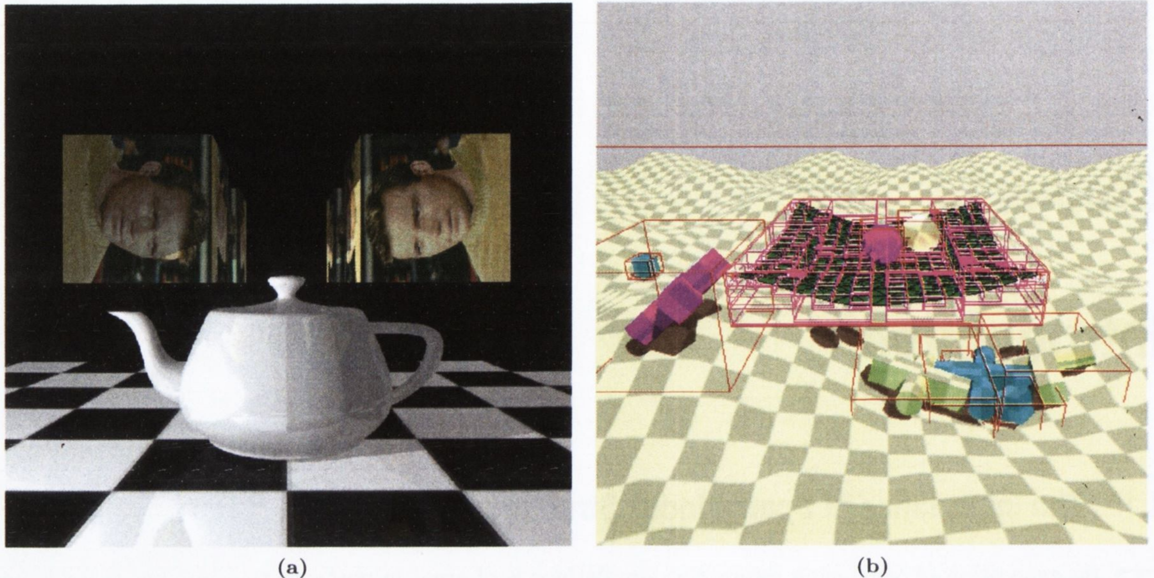
1.1 The Importance of Spatial Index Structures

THE generation of convincing interactive simulations is of great importance to a number of fields, including virtual reality, computer aided design (CAD) and video games. Such interactive simulations typically employ some form of rendering engine, along with a wide variety of other subsystems including components for physical simulation, animation, artificial intelligence, and dynamic audio processing.

Although these subsystems serve different purposes and present different challenges to the researcher or engineer, many fundamentally involve the solution to what are often elementary problems of geometry in what is usually 3D space. For example, a rendering algorithm might need to determine the visibility between two points in a scene, a collision detection routine might need to find the overlap of two objects to resolve collisions, and an artificially intelligent agent might need to find a path around several obstacles. For any complex scene, there may be millions if not billions of these elementary operations which need to be resolved. Given the ever-present drive towards greater and greater complexity in interactive simulations, naive solutions quickly become ineffective. This necessitates some mechanism for accelerating these operations.

The fundamental problem of naive solutions to these tasks is that they gather no information about the location, relative position or size of the objects of interest. Without this information, a naive solution is a brute force solution. The naive ray-tracer tests all rays against all primitives, even when the ray originates far from the primitives of interest and is travelling in the opposite direction. The naive collision detection algorithm has no notion of whether two objects are anywhere near each other, yet will spend large amounts of time on huge numbers of expensive exact collision tests.

Since many components of an interactive simulation must solve spatial problems, what is needed is a mechanism that can be quickly consulted to provide information as to the spatial structure of the input data. Such a mechanism would be highly beneficial for all kinds of spatial problems. By using information provided by such a structure, an algorithm could make much more intelligent and efficient use of compute resources to obtain the solution to a given problem. The *spatial index structure* (or *acceleration data structure*) constitutes just such a mechanism. Spatial index structures provide a spatial map over the scene geometry which may be quickly consulted to accelerate a variety of geometric search queries. This explains why spatial data structures are so fundamental to simulation technology, and are essential to any interactive simulation system. Three of the most prominent applications utilising spatial index structures are *ray-based global illumination*, *rasterisation* and *collision detection*.



(a) (b)

Figure 1.1: *Left: An image produced using the author's own interactive ray-tracer, utilising a kd-tree data structure. Right: Collision detection in the Bullet Physics engine [Bul13], visualising the BVH data structures used for collision culling.*

1.1.1 Ray-based Global Illumination

Among the most visually impressive rendering methods are *ray-based global illumination* algorithms. Examples of such algorithms include Whitted-style ray-tracing [Whi80], path-tracing [Kaj86] and photon mapping [Jen96]. The computational cost of such algorithms has been, until recently, prohibitive for interactive applications. However, a general increase in the performance of consumer hardware, the evolution of multi-core CPUs and GPGPU computing, and a number of algorithmic advances in the last decade, have brought these algorithms firmly into the realm of real-time application.

One of the key algorithmic components of this progression has been the advances made in spatial index structures [Hav00, WH06, PL10]. The primary use of spatial index structures in ray-based rendering algorithms is to accelerate the process of finding intersections between rays and scene geometry. Intersections are found by traversing the spatial index with a ray, and terminating when an intersection is found or when the spatial index has been fully searched. Only those portions of the structure that the ray intersects require further investigation, resulting in dramatically improved efficiency. The most commonly used structures for ray-based rendering are BVHs [Cla76, WHG84, WBS07, LYTM06, PL10], *kd-trees* [Ben75, SSK07, ZHWG08], and *uniform grids* [FTI86, WIK*06, KBS11]. Utilising such structures in rendering typically improves the final frames per second by several orders of magnitude. Such large performance improvements are achieved as the computational complexity of the rendering process is significantly reduced, going from an $\mathcal{O}(n)$ operation to an $\mathcal{O}(log\ n)$ operation for many widely used structures.

Aside from ray intersections, spatial index structures are useful for other parts of a ray-tracing pipeline. Some researchers have proposed constructing higher dimensional data structures containing the rays themselves [AK87], and photon mapping techniques estimate global illumination effects utilising a

spatial data structure of points [Jen96].

1.1.2 Rasterisation

The rasterisation algorithm has become the *de facto* standard rendering algorithm for interactive graphics. This can be attributed to the fact that the algorithm is relatively inexpensive compared to global illumination methods, and also because fast hardware implementations have been part of commodity GPUs since their inception. Rasterisation algorithms operate on scene primitives, projecting them onto the image plane and determining which pixels they overlap. The important notion in this process is the idea of triangle ordering. It is necessary to determine which triangles occlude others in the scene to ensure that only the closest visible triangles are drawn to the screen. Z-buffering methods, which keep a record of distance values to previously drawn triangles, are commonly used for this purpose.

As with ray-based algorithms, spatial index structures are also very useful for rasterisation methods. The BSP tree can be used to provide a front-to-back ordering of primitives from arbitrary viewpoints [HKB80, GC91]. Screen-space spatial data structures are also utilised in techniques such as irregular z-buffering, which allows arbitrary placement of samples in the image plane [JLBM05].

The drawback of rasterisation is that it cannot directly model global effects, and various workarounds are needed to approximate details such as reflections, shadows and transparency. Ray-based algorithms naturally support all of these effects and give physically accurate results. For this reason, ray-based algorithms can yield higher quality renders, while also being simpler to implement. Nevertheless, the performance of rasterisation has resulted in its continued use in interactive scenarios, and fixed-function hardware for rasterisation remains in commodity GPUs.

1.1.3 Collision Detection

Collision detection is a fundamental component of any physical simulation subsystem. The collision detection process is usually divided into two or three phases. These phases are the *broad phase*, the *narrow phase* and sometimes the *mid phase*. The purpose of the narrow phase is to take two scene primitives and determine if they are in contact. As this is a relatively expensive operation, the purpose of the broad and mid phases is to cull away as many narrow phase tests as possible beforehand, with the aim of accelerating performance.

Spatial index structures are used heavily in this process. The broad phase typically operates on entire objects, enclosing them in a bounding volume such as an axis-aligned bounding box (AABB) or sphere, or in a hierarchical structure such as an AABB tree [vdB98], oriented bounding box (OBB) tree [GLM96] or k-DOPs [KHM*98]. An inexpensive test can be used to determine if the bounding volumes of two objects overlap. If there is overlap, this information is passed on to the mid or narrow phases. If no overlap exists, no further collision checks need be performed, dramatically decreasing the workload.

The value of spatial index structures to collision detection also extends to the mid phase. The mid phase operates within objects and performs a similar task to the broad-phase. Two objects can be tested for possible collision by first building a hierarchical structure within the objects and then traversing both structures simultaneously, and following subtrees where overlap exists between the two objects. Once the leaves have been reached, a small set of potentially colliding primitives is generated and sent to the narrow phase of the collision detection system which finds precise collision points between individual primitives.

1.2 A Case for Hardware Support

The ubiquitous presence of spatial index structures in computer graphics and simulation technology, as well as performance concerns in constructing and managing such structures, has led some to speculate that specific hardware support for constructing and traversing these structures may be beneficial [WSS05, NPP*11, DFM12]. At the present time, a number of examples of specialised hardware utilising such structures exist. However, to the author's knowledge, no examples of specialised hardware for *building* such structures has ever been demonstrated in the academic literature. The goal of this thesis is to investigate in detail this question and to produce the first microarchitecture for the construction of a spatial index structure.

We can summarise the motivation for including hardware support for spatial index structures in a graphics processor or other heterogeneous computing platform as follows:

- **Performance**

  The construction of spatial index structures is a relatively expensive task for many applications, which has stimulated a great deal of research into faster construction and traversal of these structures. In general, dedicated microarchitectures can typically provide greater performance compared to software methods and this has been demonstrated in a variety of applications relevant to interactive simulation, including rasterisation [LK11], ray-tracing and global illumination [WBS06, NPP*11, Ima12], and collision detection [ZK03, RHAZ06]. Given that spatial index construction is so widely applicable, and notable precedents exist for the use of hardware acceleration for other aspects of interactive simulation, it is worthwhile to investigate if significant performance gains can be achieved with a dedicated microarchitecture for spatial index construction.

- **Efficiency**

  Aside from performance increases, a related and important metric is *performance efficiency*. Efficiency can be evaluated in a variety of ways, including performance per chip area, performance per clock cycle and performance per Watt. The question of efficiency is particularly interesting, as any simulation will have several algorithms, each contending for compute and memory resources. Even if two alternative algorithms are equal in performance, the most efficient algorithm is desirable as it frees up resources for other computations. Spatial index construction algorithms have been shown to scale well, and recent algorithms have been shown to effectively scale on large many-core platforms [Wal12, Kar12, GHFB13]. Optimum performance for spatial index construction is therefore obtained when most of the resources of the chip are dedicated to it. It is likely that a well designed custom microarchitecture could exceed the performance of many-core software implementations while using much fewer resources. This is true because a dedicated approach is inevitably much more streamlined, and the type and quantity of compute and memory resources used can be carefully tuned and partitioned for the algorithm. With such a streamlined approach, dedicating almost the entire chip to completing a single task at high performance would not be necessary, freeing up resources for other aspects of the simulation loop, thus having a performance impact above and beyond any acceleration that may be achieved for the spatial index construction itself.

- **Mobile Devices**

  Continuing advances in computing power and miniaturisation have enabled the advent of what may be termed the *mobile revolution*. Smartphones, tablet computers and mobile gaming consoles have

today reached an unprecedented level of sophistication and capability. Only ten years ago, it was almost inconceivable to think that one could have a quad core processor in their pocket, but now such devices are commonplace. The mobile revolution has brought its own goals and challenges. While mobile devices are advancing rapidly, they cannot deliver performance on par with even a budget desktop system. Additionally, power and battery life are major concerns for mobile devices. In many cases, systems designers have thus elected to include custom microarchitectures in mobile chips as a means of obtaining greater performance and also for preserving battery life. The NVIDIA Tegra mobile chips which are included in many tablets and smartphones, utilise dedicated support for HD video, audio decoding, image processing, and fixed-function rasterisation support for this very reason [TMCS08, NVI11]. Other researchers have proposed that fixed-function devices could help to enable high-quality global illumination on mobile devices [LLN*12], while remaining within the limitations of mobile platforms.

- **Power Efficiency and Dark Silicon**

  Power efficiency has now become a major concern not just for mobile devices, but also on the desktop and in the server room. Moore's law has predicted in the past that transistor counts would double every 18 months, and this continues to be the case. Power efficiency in the past has scaled in the same way, allowing larger and faster chips to remain within power budgets. However, the failure of Dennard scaling [DGR*74] has led many researchers to hypothesise that multi-core and many-core scaling will quickly become power limited in the near future. Esmaeilzadeh et al. show that at 22nm, 21% of a fixed-size chip must be powered off, and at 8nm, it could be more than 50% [EBSA*11]. This had led some to coin the term *dark silicon*, for logic which must remain idle due to power limitations. Moving forward, these events call for a new wave of power-efficient computer architectures. Some researchers [VSG*10, CMHM10] argue that efficient custom microarchitectures could help future heterogeneous single-chip processors to overcome these technology imposed utilisation limits. It is now a matter of identifying the most suitable algorithms for custom logic implementations for the age of dark silicon.

- **Flexibility**

  Flexibility is a highly desirable attribute in modern processors. Therefore, efforts to improve efficiency based on specialisation must be concious of this fact. For some stages of an application's pipeline, flexibility may be critical, whereas for others it may be less critical. To construct a highly efficient and flexible system, one must be careful of which components to specialise. Spatial index structures are typically what we may term *assistive structures*, meaning that they help to accelerate the process of obtaining a result, but do not determine that result *per se*. Taking the specific example of ray-tracing, spatial index structures are very often used to determine ray/primitive intersections. In a typical renderer, this is a well-defined mathematical problem with a unique result. Regardless of the spatial index used, whether it be a kd-tree, a BVH, or a uniform grid, the final result of a ray query will be the same. The principal difference between different structures is therefore a matter of efficiency. Given the rise of real-time ray-tracing in the past decade, future graphics processors will likely execute many such operations. If specialised hardware is provided in such systems to assist with spatial index creation or traversal, it will not place severe limits on the flexibility of the device, while providing potentially large efficiency and performance improvements.

1.3 Research Question

At the time of writing, the construction and management of spatial index structures is particularly topical in the field of ray-tracing, and a large body of recent research has been directed specifically to constructing and managing spatial index structures for this application [PL10, SBU11, Kar12, Wal12, GHFB13]. Furthermore, the recent move towards more dynamic scenes has motivated the use of object hierarchies such as the BVH as opposed to spatial partitioning structures such as kd-trees.

In this thesis, the author will focus on solving the hardware problem for ray-tracing, as many domain-specific practices are in use in this topic, and also because it represents one application for which hardware support might yield the greatest benefit. This leads to the following research question:

> Can the adoption of a custom microarchitecture for the construction of BVHs offer benefits to an interactive ray-tracing system in terms of performance and efficiency?

1.4 Contribution

The contributions of the thesis can be summarised thus: A novel microarchitecture design for fast and efficient construction of SAH-optimised bounding volume hierarchies for ray-tracing is presented. The architecture displays high performance, area and power efficiency compared to existing methods. To the author's knowledge, this architecture is the first of its kind. Alternative architectures based on this design, including a hybrid programmable/fixed-function solution, are also discussed. Applications beyond ray-tracing are considered.

1.5 Publications

The following is a list of the author's publications published over the course of this thesis.

1.5.1 Directly Relevant

- **Doyle M. J.**, Fowler C., Manzke M.: A Hardware Unit for Fast SAH-optimised BVH Construction. *ACM Trans. Graph. 32*, 4 (July 2013), 139:1-139:10

- **Doyle M. J.**, Fowler C., Manzke M.: Hardware Accelerated Construction of SAH-based Bounding Volume Hierarchies for Interactive Ray Tracing. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 209-209

1.5.2 Indirectly Relevant

- Fowler C., **Doyle M. J.**, Manzke M.: Adaptive BVH: An Evaluation of an Efficient Shared Data Structure for Interactive Simulation. In *Proceedings of the 30th Spring Conference on Computer Graphics* (Smolenice, Slovakia, 2014), SCCG '14, Eurographics Association.

- Woulfe M., **Doyle M.**, Manzke M.: Collision Detection Hardware Optimised for Ray-Tracers. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, 2010), HPG '10, Eurographics Association

- Woulfe M., Creedon E., Brennan R., **Doyle M.**, Manzke M.: Programming Models for FPGA Application Accelerators. In *Proceedings of the 1st Workshop on Programming Models for Emerging Architectures* (Barcelona, Spain, 2009), PMEA '09, Barcelona Supercomputing Center, pp. 77-83.

1.6 Summary of Chapters

This chapter has established the widespread importance of spatial index structures, as well as the motivation for the use of custom microarchitectures for improving the efficiency of such operations in future graphics processors. The remainder of this thesis consists of four additional chapters.

- Chapter 2 (Background and Related Work) begins with a brief review of some basic ray-tracing concepts, and looks at some of the most important ray-based rendering algorithms in use today. Following this, a detailed look at some of the most widely used spatial index structures used in ray-tracing is presented. Particular treatment is given to high-performance construction of these structures on modern multi-core and many-core architectures. The chapter then moves on to a thorough review of published custom microarchitectures for ray-tracing. The chapter concludes with a brief look at existing microarchitectures for related applications, as well as some relevant commercial products utilising these technologies.

- Chapter 3 (Design and Implementation) begins with an overview of a novel fixed-function microarchitecture for fast construction of SAH-based BVHs for ray-tracing. The motivation for the various algorithmic and design decisions is addressed. A detailed look at the internal microarchitecture of individual system components is then presented. Finally, the implementation of the proposed design is discussed.

- Chapter 4 (Evaluation) presents an evaluation of the proposed microarchitecture in terms of a variety of performance and efficiency metrics. The design space of the system is explored along a number of parameters. Comparison to existing multi-core and many-core implementations of equivalent algorithms is emphasised. The chapter concludes with an in-depth discussion of a number of pertinent issues, and places the proposed hardware in a wider context.

- Chapter 5 (Conclusion and Future Work) considers the obtained results in the context of the original research question. A discussion of a variety of avenues for future work is presented.

Chapter 2

Background

THE work in this thesis draws upon three major topics: *ray-based global illumination*, *spatial index structures* and *ray-tracing hardware architecture*. In this chapter, a review of the ray-tracing paradigm for rendering is presented, including a look at some of the most important ray-based rendering algorithms in use today. Secondly, the concept of the spatial index structure is examined in more detail, and in particular, a thorough assessment of the state-of-the-art of the bounding volume hierarchy (BVH) structure is given. Finally, the chapter concludes with an in-depth catalogue of existing work on ray-tracing hardware architecture.

2.1 Rendering Algorithms

The principal goal of much of computer graphics is the generation of realistic imagery. Generally, the more accurate a representation of real world phenomena that a given algorithm can produce, the more satisfactory the result. All rendering algorithms aim to reproduce the light transport that occurs in a scene, and to produce an image representing what would be seen by an observer if the scene existed in the physical world. Given that rendering algorithms all try to achieve the same goal, they can be characterised as more or less accurate solutions to what has come to be known as the *rendering equation* [Kaj86]. The rendering equation is a recursive integral equation which provides a framework within which to view all rendering algorithms as the solution to a single equation. The equation describes the intensity of light travelling from point x' to point x in a scene as the sum of the light emitted and reflected from point x' towards point x.

$$I(x, x') = g(x, x')[\epsilon(x, x') + \int_S \rho(x, x', x'')I(x', x'')dx''] \tag{2.1}$$

where $I(x, x')$ is the intensity of the light travelling from point x' to x, $g(x, x')$ is a visibility term, which will be zero if x is occluded from the perspective of x', $\epsilon(x, x')$ is the intensity of light emitted from x' towards x and $\rho(x, x', x'')$ is a term which captures the intensity of light reflected onto x by point x'' via the point x'. The integral in the equation is over every point in the scene S, and thus captures global illumination information.

Many rendering algorithms have been developed to solve this equation. Some compute only part of the equation. *Direct illumination* algorithms consider only light coming directly from light sources, while

global illumination algorithms attempt to capture indirect effects such as reflection and refraction. Even within these categories, variations exist as to how much light is accounted for.

2.2 Ray-based Global Illumination

Among the most complete methods for solving the rendering equation are those which directly simulate light transport as a means of calculating illumination at a given point. The basic concept of tracing rays through a virtual scene is the foundation of some of the most widely used global illumination algorithms. In the following sections, a description of a number of these algorithms is given. Some of these algorithms, such as *Whitted-style ray-tracing* and *Photon Mapping*, are more suited to interactive use, and are the focus of this thesis. Other algorithms, such as *Path-Tracing* and *Metropolis Light Transport*, have historically been used for high-quality offline rendering, but a description of these is also given to illustrate the pervasive influence of the ray-tracing paradigm in rendering systems. Furthermore, such algorithms are beginning to emerge in more interactive contexts [Jac13].[1]

2.2.1 Ray-Casting

The simplest ray-based rendering algorithm can be attributed to the work of Appel [App68], and may be termed *ray-casting*. Although it models only direct lighting, it illustrates well the basic concept that is at the core of all ray-based illumination algorithms. The input to the ray-casting procedure is a mathematical description of a set of geometry, some number of light sources and the position and orientation of a virtual camera relative to the scene.

Rendering an image essentially involves the projection of a three-dimensional scene onto a two-dimensional projection plane. In ray-casting, a projection plane is erected at a distance d from the virtual camera and is divided into a regular grid of elements. Each element of this grid corresponds to a pixel of the target display device. To render the image, it is necessary to determine the appropriate colour for each pixel in this grid.

To achieve this, a *ray* is cast from the virtual camera through each element of the viewing plane (Figure 2.1). The first intersection of this ray with the scene geometry is found. The point of intersection represents the point in the scene visible to the virtual camera through the current element of the projection plane.

Once the intersection point has been found, the point is *shaded*. Additional rays are cast from the point of intersection to each of the light sources. These rays are often termed *shadow rays*. Shadow rays are used to determine which light sources are visible from a given point, and thus to determine which light sources contribute to illumination at that point. Multiple light sources may exist in the scene and thus many shadow rays may be cast from a single intersection point. Once the incoming illumination from light sources is determined, the material properties of the point are used to calculate the final pixel colour. This operation is performed for all pixels in the image plane to produce the entire image.

The principle of tracing rays beginning at a virtual camera is sometimes referred to as *backwards ray-tracing*. In reality, vision is determined by light entering the eye (or other sensor) rather than being emitted from it as this method suggests.

[1] Although this thesis focuses on ray-based algorithms, other approaches to global illumination have been proposed. A commonly used example of such an algorithm is Radiosity [GTGB84]. However, ray-based algorithms remain some of the most widely used.

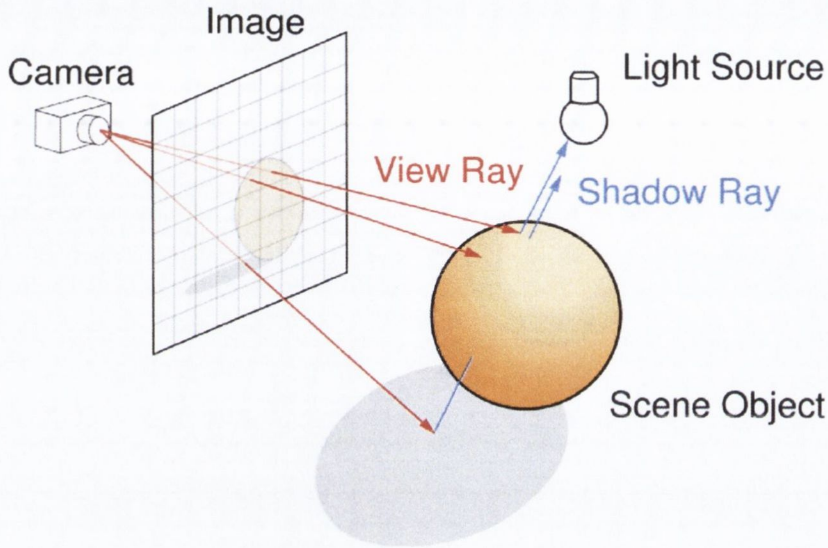


Figure 2.1: *Ray Casting.*[1]

2.2.2 Whitted Ray-Tracing

The algorithm described by Appel only accounts for light that originates from light sources. However, real-world illumination includes many indirect effects such as reflections and refraction. To accommodate a greater variety of optical phenomena, the algorithm proposed by Appel was further developed by Whitted [Whi80]. Whitted's algorithm is commonly referred to as *Whitted ray-tracing*. The algorithm is of a recursive nature. As with ray-casting, Whitted ray-tracing casts rays through the projection plane and also to light sources, but casts additional *secondary rays* originating at intersection points. Secondary rays are so called to distinguish them from *primary rays*, which are those initially cast from the virtual camera. These secondary rays capture indirect reflection and refraction illumination effects. If the intersection surface is specular, a *reflection ray* is cast into the scene. The reflection ray r is calculated in accordance with the *Law of Reflection*:

$$r = i - 2n(i \cdot n) \tag{2.2}$$

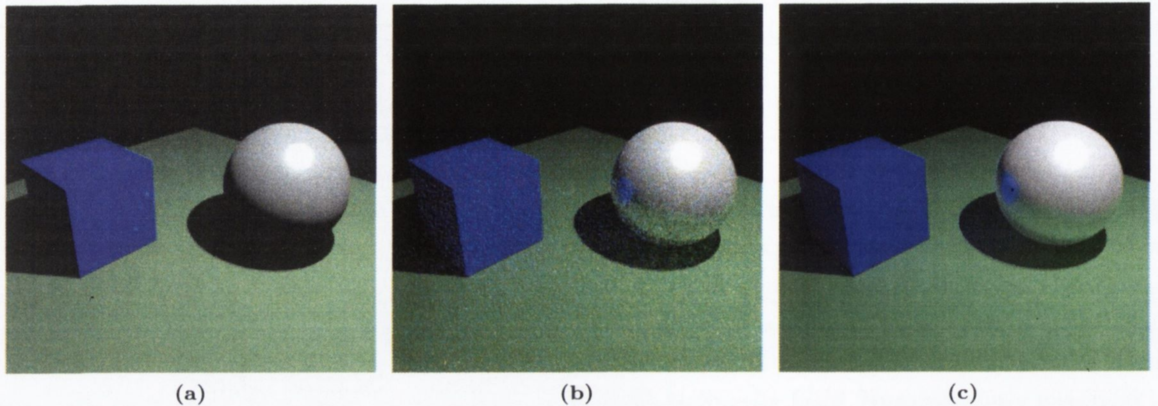
where i is a unit vector representing the incident ray, and n is a unit vector representing the normal to the surface at the point of incidence.

If the surface is transparent, a *refraction ray* is traced through the object. A refraction ray t may be calculated according to *Snell's Law*:

$$t = \frac{\eta_1}{\eta_2}i - (\frac{\eta_1}{\eta_2}cos\theta_i + \sqrt{1 - sin^2\theta_t})n \tag{2.3}$$

where η_1 and η_2 are the refractive indices of the adjacent materials at the point of intersection, θ_i and θ_t are the angles of incidence and refraction respectively, and i and n are the incidence and normal vectors respectively.

[1] Image reproduced under the *Creative Commons* license.



(a) (b) (c)

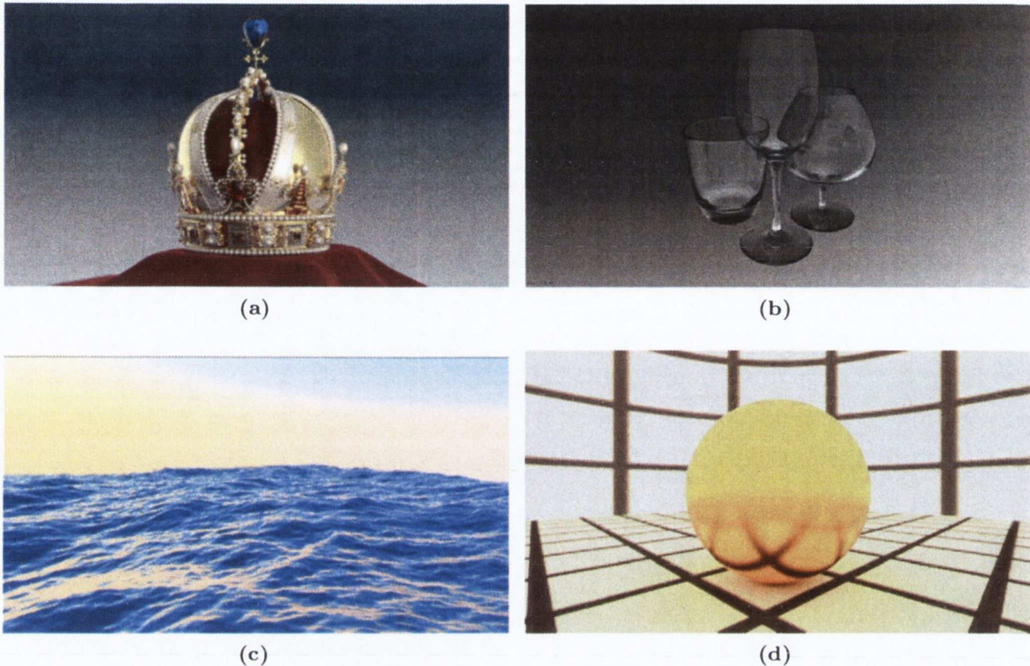
Figure 2.2: *Ray based Global Illumination: Figure (a) shows a rendering utilising only direct lighting, hard shadows and no reflections. This image is similar to what is produced with ray-casting or a Whitted-style approach. Figure (b) shows an image rendered with bidirectional path-tracing, but which has not yet fully converged to acceptable noise levels. Figure (c) shows a high quality ray-based global illumination render which has had time to converge. Reflections of the cube and the ground plane can be seen, as well as indirect lighting on the cube towards the left of the image. Images produced with LuxRender [Lux12] and an original scene.*

These secondary rays are traced though the scene in turn, recursively producing further secondary rays on intersection with scene geometry. The recursive casting of secondary rays leads to a *ray tree*, with each ray contributing to some degree to the final colour of the pixel. Allowing this algorithm to continue for several *bounces* will yield more accurate results, and could in theory continue indefinitely. In reality, when light reflects from and refracts through surfaces, it loses energy on each interaction with an object. Eventually, the energy being transmitted is small and its contribution to the illumination becomes minimal. It is thus common to limit the depth of the ray tree, either by specifying an explicit depth or by triggering termination of a ray's traversal when a large proportion of its energy has been absorbed.

The Whitted-style algorithm can be augmented through a variety of techniques which make further use of the ray-tracing concept. Such techniques include super-sampling to remove aliasing effects [Coo86], *distribution ray-tracing* [CPC84] to produce soft shadows and motion blur effects and sampling of diffuse inter-reflection for global illumination [WRC88].

2.2.3 Path-Tracing

Although Whitted ray-tracing captures some indirect illumination effects, it captures only a small fraction of light transport. More advanced algorithms such as *path-tracing* [Kaj86] attempt a more complete solution to the rendering equation. The standard path-tracing process begins by tracing rays from the camera in much the same manner as Whitted-style ray-tracing. Instead of casting shadow, reflection and refraction rays, path-tracing follows a Monte Carlo random sampling approach over the integral in the rendering equation to solve for the light transport in the scene. The rays spawned from this random sampling are traced through the scene in turn, producing more randomly sampled rays at further intersection points and so on. This process calls for a very large number of rays to be traced, but can produce very high quality images which intrinsically include global illumination effects that must be explicitly handled in other rendering algorithms. Although it is capable of generating higher quality



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

Figure 2.3: *Examples of the realistic results possible with ray-tracing methods. Images generated using Intel Embree [Ern12] and NVIDIA OptiX [PBD*10].)*

images than Whitted-style ray-tracers, the core operation of casting rays is the same, and the algorithms differ largely in how rays are generated at intersection points.

Bidirectional path-tracing [LW93, VG94, VG95] is a related algorithm which can be seen as a generalisation of regular path-tracing. The basic difference in bidirectional path-tracing is that rays are shot not only from the eye, but also from the light sources. Two random walks are performed for an image sample; one from the light source and one from the eye. Ray directions are selected via importance sampling and a Russian Roulette [AK90] termination scheme is used. When both walks have been constructed, each point along one walk is connected to each point on the other walk using shadow rays. Weighted contributions from all the points are then accumulated. The algorithm is shown to reduce to conventional path-tracing by the correct choice of weighting strategy. Bidirectional path-tracing shows significantly faster convergence compared to conventional path-tracing and thus can produce images with less noise for a given amount of work.

Metropolis Light Transport (MLT) [VG97] is a technique based on path-tracing which improves the efficiency of rendering in difficult situations such as bright indirect lighting. It achieves this by sampling light paths to a degree which is proportional to their contribution to the image. The basic algorithm begins by sampling a set of n paths using bidirectional path-tracing. Rather than continually tracing random paths, MLT continues by "mutating" the current light paths. It achieves this in a number of ways, including deleting part of a path and inserting a new part in its place. Mutations are proposed by the algorithm and then accepted or rejected based on an *acceptance probability*, which is chosen to promote paths which contribute substantially to the image. When an important region in the path space is found, the MLT algorithm can explore it locally through small path mutations, and since new paths are produced

from old ones, work can be amortised over many paths. In typical scenes, MLT is competitive with other path-tracing methods, but excels in difficult scenes where much of the illumination arises from a small set of paths. In such situations, it can achieve significantly higher quality images for the same execution time.

2.2.4 Photon Mapping

The photon mapping algorithm was developed by Jensen [Jen96]. It is a two-pass algorithm, which is capable of generating realistic global effects such as diffuse inter-reflection and caustics. The first pass of the algorithm involves the creation of the *photon map*. Photons are emitted from the light sources and followed until they meet a surface in the scene. This operation is essentially the same as tracing rays as in the other algorithms discussed here. A probabilistic algorithm is then used to decide if a photon is absorbed, reflected or refracted through the given surface. The intersection points are then stored in the photon map. The most commonly used data structure for photon maps is the kd-tree, although the BVH (Section 2.4) has also been used to good effect [FD09]. Specialised photon maps can be constructed for photons which follow certain types of light paths. One example is the *caustic map*, which only stores photons which interact with specular surfaces along their path, and can be constructed along with the global photon map. The second pass of the photon mapping algorithm involves ray-tracing the scene and estimating global illumination effects at the intersection points from the photon maps.

2.3 Spatial Index Structures

To produce high resolution, realistic images using ray-tracing, a large number of rays must be cast. To calculate intersection points, a naive ray-tracing algorithm would simply check each ray against all primitives in the scene. Given that a display may consist of several million pixels, and a scene may consist of a similar number of primitives, rendering even a single frame leads to vast amounts of computation. If interactive or real-time applications are to make use of ray-tracing, then acceleration mechanisms must be introduced. Whitted determined that approximately 75% of the running time of a naive ray-tracer is spent in intersection tests between rays and scene geometry, and that this figure can rise to as much as 95% for more complex scenes [Whi80]. It would thus be prudent to look for more sophisticated methods for finding intersection points.

Perhaps the most effective method proposed to date for accelerating intersection testing is through the used of *spatial index structures*. Spatial index structures achieve acceleration by eliminating large numbers of intersection tests from consideration by first performing inexpensive rejection tests to determine if a ray could possibly result in an intersection. Rays which do not pass this test can be discarded from consideration. The simplest form of spatial index is the *bounding volume*, or *extent* (Figure 2.4) [RW80].

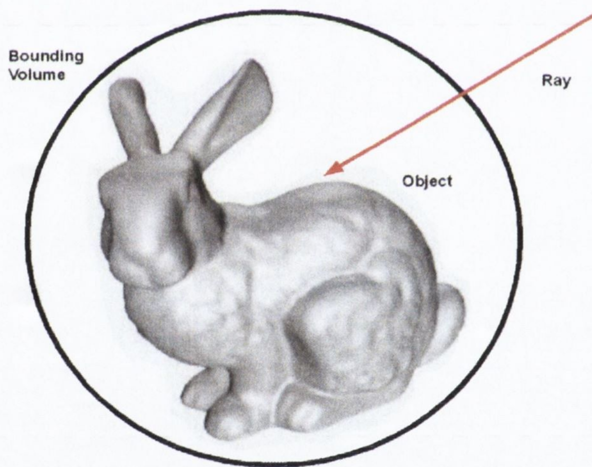


Figure 2.4: *A simple bounding volume. Intersecting the bounding sphere first can eliminate the need to check the complex primitive inside for intersections.*

The purpose of the bounding volume is to fully bound scene geometry within a simple geometric shape, such as a sphere or rectangular solid, and to test rays against this shape before checking the geometry contained within it. This provides a very low-cost way to eliminate those rays which have no possibility of intersecting the contained geometry, and thus significantly reduces the average cost per ray. All spatial index structures are based on the idea of knowing the extent of collections of scene primitives. The operation of searching a spatial index with a ray in this manner is known as *traversal.*

2.3.1 Flat vs. Hierarchical Structures

Spatial index structures can be classified as either *flat* or *hierarchical.* Flat data structures are distinguished by the property that only a single level of subdivision is present in the data structure. The bounding

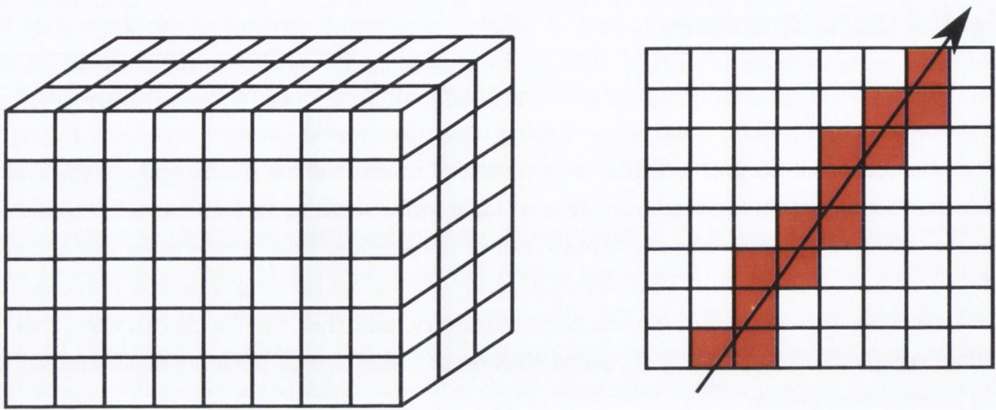


Figure 2.5: *A uniform grid in 3D (left), and a 2D representation of grid traversal for a sample ray (right). Only primitives residing in voxels which are visited by the ray (shown in red) need to be tested for intersection.*

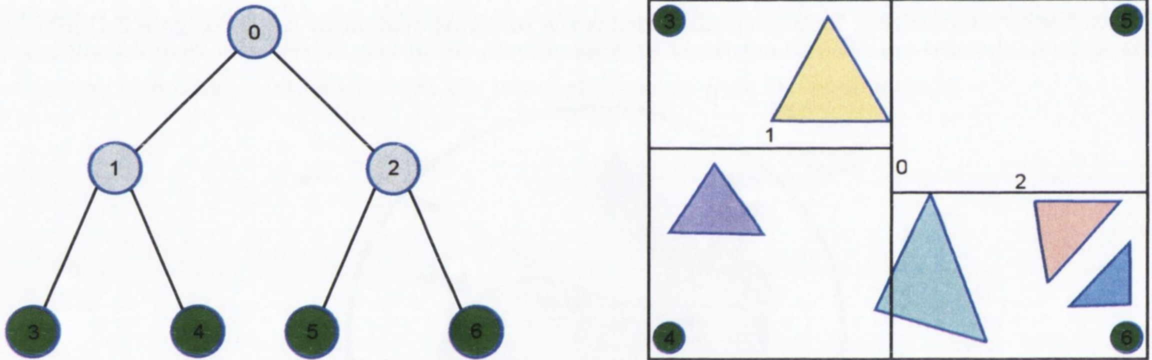


Figure 2.6: *A kd-tree showing interior nodes (0, 1, 2) in grey and leaf nodes (3, 4, 5, 6) in green. Traversal of the structure is achieved by examining the relationship of a ray with the axis-aligned splitting planes, which allows for rejecting large numbers of scene primitives as possible intersection candidates.*

sphere in Figure 2.4 would thus be classified as flat, as no further spatial information is captured beyond the outer bounding volume. A more useful example of a flat data structure in the *uniform grid* (Figure 2.5) [FTI86]. The uniform grid imposes a regular lattice of rectangular spatial regions (known as *voxels*) over the scene geometry. Decomposing a scene into multiple spatial regions in this manner provides finer spatial detail for ray traversal operations compared to a simple bounding volume.

Traversal with the uniform grid is accomplished by visiting only those voxels encountered along the ray in order (right of Figure 2.5). In this manner, only those subsets of the scene geometry which overlap this set of voxels need to be examined for intersections. For a well constructed uniform grid, this subset should be very small relative to the total number of primitives in the scene. The uniform grid is classified as a flat structure, as once a ray enters a voxel, no further spatial information is provided as to the organisation of the contained primitives, and thus the ray must intersect all primitives residing within the voxel to be sure of finding all ray intersections.

Most spatial index structures used in ray-tracing today are *hierarchical*. In a hierarchical data structure, subdivision of the scene is accomplished in multiple levels, forming a tree structure. The *kd-tree* [Ben75] represents the classic example of a hierarchical spatial index (Figure 2.6).

The classic kd-tree is a binary tree which subdivides space using a hierarchy of splitting planes. Each interior node in the kd-tree represents an axis-aligned splitting plane, and each leaf node represents a spatial region delineated by those planes. Figure 2.6 shows the correspondence between nodes in the kd-tree hierarchy with splitting planes and spatial regions.

Traversal of hierarchical data structures is well exemplified by the kd-tree. Traversal begins at the root node of the hierarchy. By intersecting the ray with the root splitting plane (plane 0 in Figure 2.6), it is possible to determine which of the two spatial regions delineated by this plane must be further examined for intersection. If the ray crosses the splitting plane within the scene boundary, then both children of the root must be visited. If the ray remains on one side of the plane as it travels through the scene, then all primitives residing on the other side can be safely rejected as intersection candidates. To complete traversal, this procedure is repeated for descendants of the root until a leaf node is reached. At which point, all primitives referenced in that leaf are tested for intersection with the ray.

The principal advantage of hierarchical structures such as the kd-tree is that they allow for adaptively subdividing regions where additional spatial information would be beneficial, such as very dense areas of the scene. In contrast, flat structures such as the uniform grid provide uniform subdivision over the scene and cannot adapt to uneven geometry distributions, leading to a less efficient data structure.

2.3.2 Spatial Subdivision vs. Object Partitioning

Spatial index structures are typically categorised as either *spatial subdivision* structures or *object partitioning* structures. The kd-tree and the uniform grid both represent examples of spatial subdivision structures. Spatial subdivision structures organise space itself into sub-volumes, with no strict regard for the extent of constituent scene primitives. In such a structure, a primitive may overlap the division between several spatial regions, and can be referenced in multiple hierarchy nodes. Figure 2.6 shows that this property holds true for kd-tree.

Object partitioning structures represent the second major category of spatial index. Examples of such structures include the bounding interval hierarchy (BIH) [WK06] and the bounding volume hierarchy [Cla76]. Object partitioning strategies can be considered the dual of spatial subdivision structures, in that geometry itself is the subject of partitioning rather than space. Nodes in an object partitioning structure represent the extent of groups of primitives, rather than spatial regions *per se*. The consequence of this approach is that the spatial regions encompassed by nodes may themselves overlap.

Early work on interactive ray-tracing established spatial partitioning structures such as kd-trees as the structure of choice for static scenes [Hav00]. However, the move to dynamic scenes has motivated the increased use of object partitioning strategies, as dynamically updating these structures is more easily achieved as they do not strictly subdivide space, but rather collections of primitives. By far the most widely used structure in this category has been the BVH, and indeed, the BVH is likely the most widely used spatial index in real-time ray-tracing today. A detailed treatment of the BVH data structure is now given, as it is central to the work of this thesis.

2.4 Bounding Volume Hierarchies

The BVH was first proposed by Clark [Cla76]. The classical BVH is a tree structure, in which each node in the hierarchy represents a bounding volume such as a sphere or rectangular solid. Weghorst et al. [WHG84] experimented with the use of different bounding volume types within the same hierarchy, and discussed the importance of tightness of fit for achieving maximum traversal efficiency. For ray-tracing applications, the vast majority of implementations utilise AABBs as the bounding volume, as simple and efficient ray/AABB intersection algorithms are known [KK86, Smi98, WBMS05].

An example BVH data structure is shown in Figure 2.7. Each leaf in the BVH (shown in green) represents the AABB of some subset of scene primitives. In the basic BVH structure, these subsets are disjoint. A reference to this subset is stored in the leaf, as well as its AABB. Interior nodes in the BVH store an AABB representing the union of all AABBs of their descendant nodes. Each interior node also stores pointers to its two child nodes.

Although BVHs have historically been binary trees, some more recent implementations have argued for generalising BVHs to higher branching factors of four or more children per node [EG08, WBB08, GL10], which allows for more efficient use of SIMD hardware in the absence of high ray coherence (the property of multiple rays to have similar origins and directions).

Classical BVH traversal is accomplished as a recursive procedure. Rays are first intersected with the root bounding volume (represented as node 0 in the hierarchy of Figure 2.7). If the ray does not intersect the root node, then it misses the scene entirely, and tracing continues with the next ray in the queue. If the ray does intersect the root volume, one child node (often the right child) is pushed to a stack, and traversal continues with the left child. Traversal continues by repeating the same intersection process for the AABB stored in the left child, and so on until a leaf node is reached. At this point, the ray is intersected with the primitives referenced within the leaf. After processing a leaf node, traversal can continue by popping a node from the stack, or terminating the traversal if the stack is empty. If at any given node, the ray does not intersect the AABB stored in that node, all descendants of the node can be safely discarded from consideration.

Modern high-performance ray-tracers often employ more advanced traversal methods, such as *packet tracing* [WSBW01, GPSS07] or *frustum culling* [RSH05, ORM08]. These algorithms characteristically gather rays into coherent groups and traverse these groups with the data structure as opposed to traversing individual rays. In doing so, it is possible to amortise traversal operations over many rays, and to more readily take advantage of SIMD operations commonly found in today's processors. However, the basic premise and flow of the traversal remains very similar. Furthermore, such methods generally rely on ray coherence, and may not provide benefit for very incoherent ray distributions.

2.4.1 BVH Construction

Constructing efficient BVHs for ray-tracing is a precise process. As any possible permutation of assigning scene primitives to leaf nodes is valid for the BVH, this leads to an extraordinarily large number of possible BVHs for any substantial scene. The strategy adopted for construction can have a profound effect on the efficiency of rendering. This necessitates a sound strategy for constructing the hierarchy.

For static scenes, BVH construction need only be performed once at the beginning of the simulation. In this case, construction of the BVH is amortised as a pre-process to rendering. For dynamic scenes, the BVH must be constantly regenerated in order for the spatial map provided by the structure to remain valid from frame-to-frame. This leads to significant per-frame overhead in the renderer. In recent times,

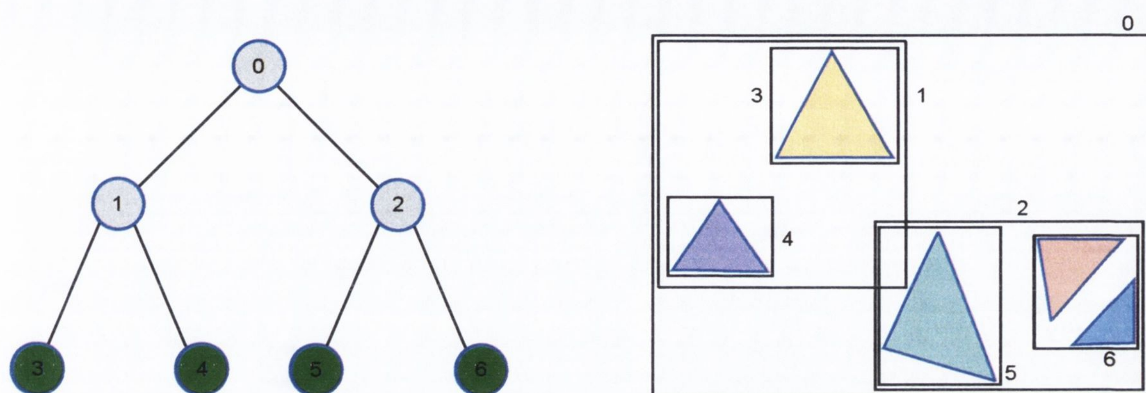


Figure 2.7: *A BVH showing inner nodes (0, 1, 2) in grey and leaf nodes (3, 4, 5, 6) in green. The BVH is the quintessential example of an object partitioning structure, where groups of primitives are the subject of subdivision. Contrasting with Figure 2.6 highlights the fundamental difference between spatial subdivision and object partitioning approaches.*

the focus of real-time ray-tracing research has increasingly shifted to addressing the challenges posed by rendering dynamic geometry. As a result, the significant cost of spatial index construction has motivated the development of a wide variety of efficient construction algorithms.

BVH construction algorithms for ray-tracing may be categorised as belonging to one of the following classes:

- ***Top-down*** The BVH is constructed by recursively splitting nodes into two child nodes, beginning at the root of the hierarchy and proceeding to the leaf nodes.

- ***Bottom-Up*** BVH construction begins by first constructing the leaf nodes, and grouping nodes into increasingly larger clusters until a root node is formed.

- ***LBVH-style*** The BVH construction procedure is rephrased as a highly parallel sorting operation. The BVH is then extracted from a sorted list of primitives.

- ***Refinement-based*** A low cost, low-quality BVH is first constructed, and one or more rounds of refinement are performed to arrive at a higher quality tree.

A fifth class of construction algorithms, *Insertion-based* builders, construct the BVH by inserting primitives one at a time into a partial tree such that the expected tree cost is minimised on each insertion. Insertion-based builders have seen very little use in recent times, and are not known to be effective for interactive applications, and so the author will limit discussion to the other four categories. Notably, however, some of the earliest construction algorithms were based on this idea [GS87].

The large per frame overhead of BVH construction has also motivated the complementary approach of *BVH updating* mechanisms. BVH updating methods are used in combination with full construction methods, and take advantage of frame-to-frame geometry coherence as a means of more cheaply procuring a valid BVH for some frames in the animation. These related methods of BVH regeneration are of particular relevance to interactive contexts, and so are also discussed here.

Algorithm 1 Generalised pseudocode for top-down recursive BVH construction.

```
1:  buildBVH(nodeN)
2:
3:  if terminationCondition(N) then
4:      makeLeaf(N)
5:      return
6:  else
7:      bestCost ← ∞
8:
9:      for all candidate partitions P of N do
10:         if expectedCost(P) < bestCost then
11:             bestCost ← expectedCost(P)
12:             bestPartition ← P
13:         end if
14:     end for
15:
16:     partitionNode(N, bestPartition)
17:     buildBVH(N.leftChild)
18:     buildBVH(N.rightChild)
19:     return
20: end if
```

Top-down Builders

For ray-tracing, many BVH construction algorithms follow a *top-down* procedure. The root node of the hierarchy is first constructed by computing the AABB of all scene primitives and storing this in the root. To construct multiple levels of the BVH, nodes beginning at the root may be split by assigning primitives referenced in the node into one of two new child nodes. For each new child node created, a tight AABB is calculated for the subset of primitives assigned to it. Hierarchy construction proceeds to the leaf nodes by recursively splitting descendants of the root in the same manner, until a termination criterion is met. Termination criteria can be based on a variety of factors, including the number of primitives referenced by a node, the depth of the node in the hierarchy, or predictions based on various cost formulae.

For each node to be split, the choice of how to assign primitives to the new child nodes can have a profound effect on rendering efficiency. Some of the simplest heuristics include calculating the *spatial median* of the node AABB along a given axis, and assigning primitives to child nodes based on which side of the median they reside. A similar partitioning strategy is the *object median*, in which primitives are sorted along a given axis, and the centre of the sorted list is chosen as a pivot point for assigning primitives to the child nodes. Algorithm 1 shows generalised pseudocode for the top-down construction method, with lines 9 to 14 representing the splitting heuristic.

Perhaps the most widely used strategy for splitting nodes in the BVH is the surface area heuristic (SAH), which was first proposed by McDonald and Booth [MB90]. The SAH estimates the expected ray traversal cost C for a given geometry partitioning, and can be written as:

$$C(V \rightarrow \{L, R\}) = K_T + K_I \left(\frac{SA(V_L)}{SA(V)} N_L + \frac{SA(V_R)}{SA(V)} N_R \right) \tag{2.4}$$

where V is the original volume, V_L and V_R are the subvolumes of the left and right child nodes, N_L and N_R are the number of primitives in the left and right child nodes, and SA is the surface area.

K_I and K_T are implementation-specific constants representing the cost of ray/primitive intersection and node traversal respectively. The SAH takes into account the geometric probability of a ray intersecting the resulting child nodes based on their surface area, as well as the cost of ray intersection with the child nodes based on the number of primitives residing within them. This widely used formula for the SAH utilises only an approximation of the true subtree cost of the child nodes, as it assumes that they will become leaf nodes, which will likely not be true for many SAH splits over the course of hierarchy construction. Nevertheless, the SAH gives very good results in practice, and is known to be among the best known construction heuristics for ray-tracing.

In a top-down builder, the SAH can be evaluated for a number of split candidates and the best candidate chosen. A number of strategies have been proposed for generating candidate partitions. *Sweep builders* sort all primitives along a given axis and evaluate the SAH for each possible sorted primitive partitioning, which yields highly efficient trees. For this reason, SAH sweep builders remain the *de facto* quality benchmark for ray-tracing hierarchies. The downside of the sweep build is that it is very costly to construct due to a large number of heuristic evaluations. Construction times on modern machines are on the order of seconds or hundreds of milliseconds for scenes featuring more than a few hundred thousand primitives.

Given that the sweep build approach to top-down construction is particularly time-consuming, a number of fast approximations have been developed with the goal of improving construction times. One widely used approximation is the *Binned SAH*, and indeed the binned SAH method is particularly relevant to the work in this thesis. Instead of evaluating the SAH formula at each primitive along a given axis, the binned SAH method selects only a small number of such candidates that are spread evenly over the candidate range. Primitives are first bucketed (or *binned*) by the their location along an axis. This forms a number of SAH *bins*, which are represented by an AABB which corresponds to the union of all primitive AABBs in the bin, and a primitive count representing the number of primitives in the bin. This information can be obtained with a single pass though the primitives, eliminating the need for a full sort of the list. By constructing such bins, it is possible to evaluate Equation 2.4 for this limited number of candidate positions. Fortunately, it has been established that, even for relatively large scenes, a small number of bins (typically 8-32) is in many cases sufficient to achieve almost the same rendering efficiency as a full sweep build, while providing construction times which are an order of magnitude faster [Wal07, LGS*09]. Figure 2.8 illustrates the binned SAH sampling process.

A number of high-performance top-down BVH construction algorithms have been proposed in recent years, mostly utilising the binned SAH method and targeted towards modern multi-core and many-core architectures. Since nodes in the BVH represent disjoint subsets of primitives, multiple nodes may freely be split in parallel. Furthermore, given that many operations within a node are independent, parallelisation can be achieved within nodes also. The proliferation of multi-core and many-core architectures has prompted the investigation of many parallel algorithms for top-down construction of BVHs. Recent implementations have utilised many forms of parallelism, including assigning subtrees to individual cores (which corresponds to spreading calls to $buildBVH$ in Algorithm 1 to multiple processing elements) and building single nodes using multiple cores (which corresponds to parallelising lines 7 to 16 of Algorithm 1). Most such algorithms have made good use of SIMD datapaths found on modern machines.

Multi-core CPU construction of binned SAH BVHs was demonstrated by Wald [Wal07]. The algorithm distinguishes between the upper and lower nodes in the tree, utilising a more data parallel approach for the upper nodes in which each core is assigned a subset of the scene primitives and is responsible for assigning those primitives to their correct SAH bin. For lower nodes, a task-parallel per

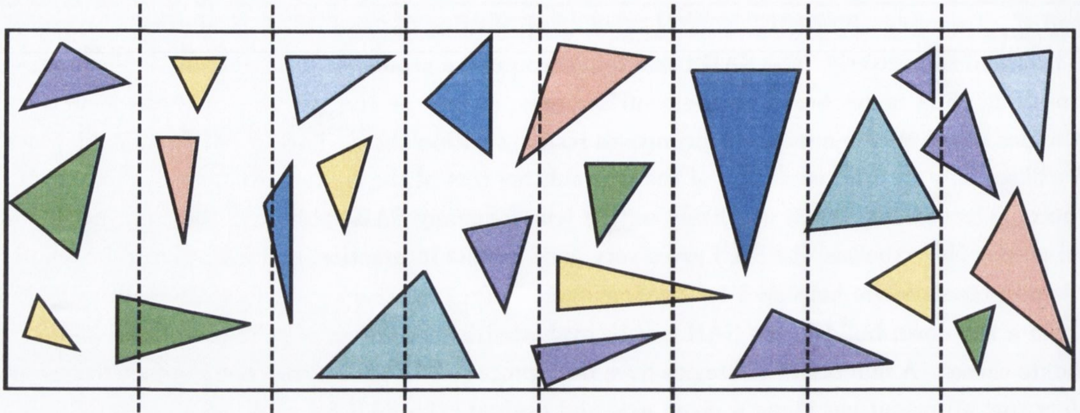


Figure 2.8: *Illustration of binned SAH sampling. Primitives (triangles in this case) are first placed in their correct SAH bin, according to the centre of their AABB or similar positional attribute. Once formed, these bins represent all the necessary information for evaluating the SAH formula at the bin boundaries (shown as dotted lines). Eight bins are pictured, allowing for seven splits to be evaluated. Generally, the split candidate resulting in the lowest SAH cost is chosen as the partitioning plane.*

subtree scheme is used, where each core independently constructs its own set of subtrees.

Lauterbach et al. describe a parallel breadth-first scheme for binned SAH construction for GPUs [LGS*09]. Each new child node produced generates a new thread in the build process, allowing for a large number of concurrent threads to effectively engage the GPU architecture. The drawback of this approach is that an insufficient number of nodes are present at the top of the tree to fully engage the GPU, and thus parallel resources are under utilised in an early stage of the computation.

Another fast GPU approach to binned SAH BVHs is demonstrated by Sopin et al. [SBU11]. The algorithm is implemented in OpenCL on an NVIDIA GTX 480 and distinguishes between three different node sizes during processing as *Large*, *Secondary* and *Small* nodes. Large nodes are those nodes containing greater than 32k triangles, Small nodes contain fewer than 256 triangles, and Secondary nodes are those that lie in between. At the Large node stage, all cores of the GPU cooperate to split the node in parallel. At the secondary stage, multiple OpenCL *work groups* are assigned to each node, with many nodes processed in parallel. At the Small stage, a single workgroup is assigned to each node. Construction times on the order of tens of milliseconds are observed, using 32 SAH bins and medium-sized scenes.

A further interesting implementation of parallel binned SAH BVH construction is demonstrated by Wald [Wal12], and is targeted toward the Intel Many Integrated Core (MIC) architecture. The Intel MIC architecture in this paper is simulated and consists of thirty-two x86 cores, each operating at a speed of 1 GHz. From an algorithmic perspective, this implementation resembles an earlier CPU BVH construction method [Wal07]. The system controls work distribution by allowing threads to request work from a task queue. The system splits build tasks into two kinds, "standard" binning tasks (used for larger nodes), and "local" binning tasks (for nodes with fewer than 512 primitives). In a standard binning tasks, a data parallel approach is used, and the primitives are partitioned into blocks of 512 each. Each thread is responsible for assigning the contents of its block to the correct SAH bin, and results from multiple threads are merged to split the node. For local build tasks, a single thread builds a full subtree. The implementation takes advantage of the MIC architecture by utilising wide SIMD instructions during binning operations. In addition, data quantisation of primitive coordinates is employed to improve cache performance.

Garanzha et al. [GPBG11] present a method for fast SAH-optimised BVH construction on GPUs. This builder is not a conventional binned SAH builder, as it uses a pre-built grid to guide construction, and generally results in slightly lower quality hierarchies than those produced with conventional binned SAH algorithms. On the other hand, it exhibits significantly higher construction speed, as the technique is capable of building a tree in $\mathcal{O}(k * n)$ in the number of primitives. The primary feature of this builder is to perform all SAH binning in a single sweep, by binning all triangles into a very fine grid in advance. For each triangle, the cell in the auxiliary grid in which its centroid lies is computed, and the resulting list of cell IDs for each triangle is sorted with a compress-sort-decompress technique [GL10]. Several detail levels of the grid are constructed, with the coarsest grid at 8^3 voxels. Grid voxels are assumed to approximate the AABB of their contained triangles. Long thin triangles are split into multiple references in advance to ensure this assumption is reasonable. Constructing the hierarchy is performed in a breadth-first manner, and begins by examining the coarsest grid and evaluating the SAH along the voxel boundaries in this grid. When a split is chosen, the node is subdivided and its children are added to a build queue. As deeper levels of the tree are computed, the finer grid detail levels are used to ensure that at least four split candidates per axis are available during split operations. Once the approximate hierarchy over the grid voxels is built, the true hierarchy is computed by using the true AABBs of the primitives to determine tight AABBs for the hierarchy nodes.

The Linear Bounding Volume Hierarchy (LBVH)

A relatively recent and important development in fast construction of BVHs is the linear bounding volume hierarchies (LBVHs) method [LGS*09]. The original LBVH method rephrases BVH construction as a highly parallel radix sort, and for this reason is very suitable for many-core platforms such as GPUs. The LBVH method first assigns $3k$-bit *Morton codes* (integer keys) to scene primitives by binning them into a regular $2^k \times 2^k \times 2^k$ virtual grid imposed over the scene. Using a k-bit code per axis, each Morton code is constructed by interleaving three codes. The list of Morton codes is then sorted by parallel radix sort. The interleaved assignment of codes results in a list of primitives which follows a Morton curve in the scene. A BVH can then be extracted from this list of Morton codes by moving through all bits of the Morton codes in order, and splicing the list at positions where two consecutive codes differ in the current bit under examination. Splicing the list in this way essentially corresponds to bucketing primitives according to the original regular virtual grid, and thus produces a BVH very similar to what would be produced with a top-down spatial median build. Figure 2.9 illustrates the ordering of Morton codes in the implicit grid.

Construction times for recent LBVH implementations on modern hardware can be less than twenty milliseconds for scenes containing up to one million primitives. Although extremely fast to construct, the LBVH method almost always results in lower quality hierarchies compared to SAH driven methods, as it does not employ such high-quality build heuristics. To improve tree quality, it is possible to combine the LBVH method with more conventional top-down construction algorithms. Lauterbach et al. proposed to construct the upper levels of the hierarchy with the LBVH method, where insufficient parallelism was present for their top-down SAH builder to be efficient on the GPU, and switched to the SAH method once a sufficient number of tasks were made available by the LBVH [LGS*09].

The LBVH concept can be extended to the hierarchical linear bounding volume hierarchy (HLBVH) [PL10], which improves construction performance by taking advantage of data coherence assumed to be present in the input mesh. The HLBVH builds a hierarchy by first sorting primitives into coarse voxels using a compress-sort-decompress radix sort, and then sorts within each coarse voxel, instead of performing

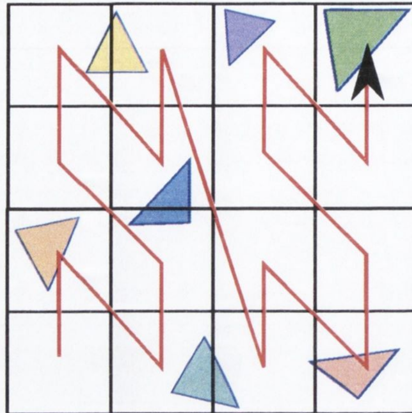


Figure 2.9: *Illustration showing the Morton curve ordering of implicit grid elements in the LBVH construction algorithm. By sorting primitives along the curve and splicing the sorted list, it is possible to construct a hierarchy closely resembling a spatial median top-down recursive build at a much lower construction cost and in a manner highly suited to many-core architectures.*

a global radix sort as in the original LBVH [LGS*09]. This work also proposes a hybrid builder similar to Lauterbach et al., but uses the SAH for the upper levels of the tree (with a full sweep build) as opposed to the lower levels, using the actual SAH cost of the lower subtrees (which are built first) as a guide for construction. The SAH construction is performed on a host CPU. The HLBVH is shown to be around three to four times faster than the original LBVH and consumes significantly less memory bandwidth. Garanzha et al. further improved the HLBVH, by moving the SAH construction to the GPU and using a binned SAH builder rather than a sweep builder [GPM11]. This implementation is approximately one order of magnitude faster than the original HLBVH.

A further improvement of LBVH type techniques was presented by Karras [Kar12]. This algorithm is motivated by the observation that earlier LBVH algorithms build the tree in a sequential fashion by processing each level in turn. This limits the amount of parallelism available in the construction and can inhibit parallel scalability. To overcome this limitation, Karras presents a method of constructing LBVH-style trees which builds levels of the tree in parallel. The algorithm maintains one array each for internal nodes, leaves and Morton code keys (which are sorted in Morton curve order). The construction is based on ordering all nodes of the tree in the nodes array such that the index of a given node in this array corresponds either to the first or to the last index of its corresponding range of Morton codes in the key array. A thread is spawned simultaneously for each node in the tree. These threads determine the range of primitives that corresponds to each node by examining the neighbouring groups and the current node's Morton codes. Once this range is determined, a binary search is performed to find the split position in this node, based on the Morton codes residing within it. Finally, the algorithm outputs the child pointers for the node. Results show that this method of constructing LBVHs results in significantly higher construction performance, as well as greater scalability on highly parallel architectures. The work also describes how the method can be adapted for constructing kd-trees and octrees of points.

Bottom-Up Builders

Bottom-up construction of BVHs is usually achieved by the *agglomerative clustering* solution to hierarchy construction. Agglomerative clustering is achieved by grouping scene primitives into successively larger

Algorithm 2 Pseudo-code for the naive agglomerative clustering algorithm.

1: agglomerativeCluster(set S of scene primitives)
2: $clusterList \leftarrow S$
3: **while** $clusterList.length() > 1$ **do**
4: $bestCost \leftarrow \infty$
5: **for all** A in clusterList **do**
6: **for all** B in clusterList **do**
7: **if** $A \neq B \wedge distance(A, B) < bestCost$ **then**
8: $bestCost \leftarrow distance(A, B)$
9: $bestPair \leftarrow (A, B)$
10: **end if**
11: **end for**
12: **end for**
13: clusterList.remove(A)
14: clusterList.remove(B)
15: clusterList.add(bestPair)
16: **end while**

clusters until the root node of the hierarchy is formed. At the beginning of construction, primitives are considered to be clusters of size one. A fundamental component of agglomerative clustering is an arbitrarily defined cluster *distance* or *similarity* function which assigns a score to each potential cluster grouping. For each iteration of the clustering, the pair of clusters with the highest similarity is found, and combined into a single new cluster. Similarity functions are often based on properties such as the distance between the two clusters, or geometric properties of the newly formed cluster, such as surface area. Although more sophisticated agglomerative clustering algorithms for achieving the same result are available, the naive agglomerative clustering algorithm conveys the basic premise quite clearly. Pseudocode is shown in Algorithm 2.

More sophisticated heap-based methods utilise a min-heap to keep track of the closest pair of primitives for every iteration of clustering, and a kd-tree for accelerating the search for the best match for each cluster.

Walter et al. applied the concept of agglomerative clustering to BVHs for ray-tracing, and proposed a new *locally-ordered* agglomerative clustering algorithm which is much more efficient than earlier naive and heap-based methods to clustering [WBKP08]. Naive and heap-based solutions to the clustering problem group the closest two clusters on each iteration. Successive iterations of clustering can thus result in highly incoherent accesses to scene clusters. The insight of locally-ordered agglomerative clustering is that individual clustering operations may be reordered while still producing the same hierarchy. If two clusters both agree on the other being their best match, then the agglomerative clustering solution is guaranteed to combine these clusters eventually, meaning that it is safe to combine the clusters at any point. To take advantage of this, the locally-ordered algorithm begins by first building a kd-tree over the clusters. One cluster, A, is selected and its closest match, B, is found. If A and B both agree that they are each other's closest match, then they can be safely combined. If they do not agree, then the algorithm proceeds with cluster B and its true closest match, performing the same test as was performed with A and B. Since subsequent clustering operations are based on finding the closest match of one of the clusters which was examined in the previous iteration, closest match search operations are much more coherent. An additional advantage is that the algorithm does not rely on a method for finding the globally closest pair in the cluster list at a given time, removing the need for a heap or other structure. These

two properties allow the locally-ordered algorithm to outperform earlier methods by a factor of 2-3×. Walter et al. applied this technique to BVHs for ray-tracing, by employing a distance function based on the resulting surface area of combined clusters. Their results show that this method can often achieve significantly higher quality than top-down binned SAH methods, but at a much higher construction cost.

Based on the work of Walter et al., Gu et al. have recently proposed a high-performance algorithm based on agglomerative clustering [GHFB13]. This algorithm, named *Approximate Agglomerative Clustering* (AAC), approximates the agglomerative clustering solution to achieve interactive rebuild speeds. To achieve this, AAC limits the clustering search space to a smaller subset of the scene at any given time. The first step in AAC is to construct an initial, implicit LBVH called the *constraint tree*. Leaf nodes in this tree consist of a small number of scene primitives, and are considered as *singleton clusters*. The next step is to recursively traverse the constraint tree top-down, until a region encompassing fewer than a threshold number of clusters is found. Once a suitable region is found, an agglomerative clustering step is performed on the subset of scene clusters. Clustering continues until the number of clusters is reduced to a certain amount, controlled by a *cluster count reduction function*. Depending on this function, this is typically several, rather than a single cluster. The result of this clustering phase is then passed back up the constraint tree. As newly clustered subsets are passed back up the tree, they are combined with other subsets which share the same parent nodes, forming a larger subset. Clustering is performed on these newly combined subsets all the way to the root node of the constraint tree. If the cluster reduction function is chosen such that more than one cluster is returned at each step, a final clustering stage is performed at the root node of the constraint tree to complete the hierarchy. Results indicate that very high construction performance (on the order of a few tens of milliseconds for complex scenes), as well as good scalability can be achieved with this method on multi-core CPUs. Furthermore, the algorithm is generally capable of constructing trees of higher quality than SAH sweep builds.

Refinement-based Builders

The development of LBVH techniques has given rise to a further, and very recent approach to BVH construction, which we may term *refinement-based construction*. The basic premise of refinement-based construction is to first construct a low-cost, low-quality hierarchy using a technique such as HLBVH, and to use the ordering information in this hierarchy to assist in the construction of a higher quality tree.

Karras and Aila demonstrate one such approach [KA13].The strategy of this work is to first build a low-quality BVH of the scene using the highly-parallel LBVH builder presented in an earlier publication [Kar12]. The next step in the algorithm is to perform several rounds of *tree optimisation* to transform the low-quality LBVH into a high-quality tree. Finally, a post-processing step is run to collapse small leaf nodes if it is expected to further improve tree quality. The optimisation stage is based on finding a number of *treelets* (small subtrees of fixed size within the full hierarchy) and optimising their topology to yield an optimal expected ray-tracing cost. The optimisation step first performs a parallel bottom-up traversal of the BVH starting at the leaf nodes. When a node is encountered that has a sufficient number of descendants such that it can form the root of a treelet, the treelet is formed and the optimisation proper begins. The optimisation works by using a dynamic programming method to exhaustively test every possible topology of the treelet, and then reinserting the best topology back into the tree. As treelets are chosen to be of a small size (around seven nodes), such an exhaustive search is feasible. The search for treelets continues to the root of the hierarchy, with a treelet being formed for every node with a sufficient number of descendants. In addition, the authors introduce a novel triangle splitting strategy to further

improve performance for scenes with non-uniform triangle sizes. Before the initial construction, a fixed split budget is first chosen, and then triangles are assigned a splitting priority, which takes into account whether they will overlap splitting planes which occur early in the initial LBVH, and also if there is a good potential for reducing the surface area of their bounding AABBs by splitting them into multiple AABBs (which may represent a tighter fit). The split budget is then distributed over the triangle set according to their priority. Rather than splitting triangles individually as earlier methods have done [EG07], the idea is to split a large number of triangles by the same splitting plane, as this has the greatest potential for reducing node overlap. To select the splitting planes, the authors utilise the planes defined by the initial LBVH, splitting triangles by the plane nearest to the root of the tree that it overlaps. Results show that trees constructed with this method offer rendering performance comparable to SAH sweep builds in most cases, while exhibiting construction performance on the order of tens of milliseconds for complex scenes. A further interesting aspect of this work is that the authors evaluate in detail the impact of the total time to image of their novel build strategy, showing that for a wide range of ray budgets, their total rendering times can be be highly competitive for many applications.

BVH Updating

The BVH data structure has enjoyed wide use in interactive ray-tracing systems in recent times. One reason why this is true is because a number of *update* methods for BVHs which can be used in combination with full rebuilds have been devised. The simplest form of updating mechanism is *refitting*. For deformable scenes where no geometry is added or subtracted, it is possible to preserve the geometry groupings expressed by the BVH of a previous frame, and to merely update the node AABBs to reflect new primitive positions in the scene. This approach is much cheaper than reconstructing a high quality BVH, but with the disadvantage that very inefficient hierarchies may be produced if the scene is highly dynamic or chaotic. However, if the trade-off of higher BVH construction speed and lower traversal efficiency is a net win, then these methods can be useful. In such cases where refitting results in lower rendering performance, full rebuilds can be periodically performed.

Laterbach et al. [LYTM06] present a simple strategy for refitting BVHs. The method copes with dynamic scene changes by refitting the BVH using a bottom-up approach. First, the bounding AABBs of hierarchy leaves are updated to reflect the new extent of their referenced primitives. The refitting process is completed by propagating the new extents of the leaves up the tree towards the root, and adjusting the AABBs of interior nodes to reflect the extents of their descendants. Lauterbach et al. proposed a degradation heuristic to detect if the BVH becomes too ill-conditioned through repeated applications of this refitting method, and a full rebuild is performed when a quality degradation threshold has been met.

Selective restructuring of BVHs has also been proposed [YCM07]. The notion of selective restructuring involves rebuilding only certain subtrees in the hierarchy and refitting others. To identify subtrees for rebuilding, a heuristic based on estimating the degree of subtree degradation (*CM-IQ*) and a heuristic for predicting the benefit of rebuilding the subtree (*RM-IQ*) are employed. The algorithm begins by refitting the hierarchy after each frame and traversing the tree to identify pairs of nodes which overlap to a significant degree. By applying the two heuristics, it is decided whether to rebuild the subtree encompassing the union of these two nodes. The technique is combined with a lazy evaluation strategy and overall produces significant improvements in total time to image versus full hierarchy reconstruction based on top-down build methods.

The concept of tree rotations was introduced to ray-tracing acceleration structures by Kensler

[Ken08]. Originally conceived as a method of optimising the tree cost of a pre-built BVH, tree rotations are based on the observation that in an existing BVH, the cost of individual subtrees can be more precisely estimated than is possible during construction. Using this information, child nodes representing the root nodes of a subtree can be exchanged between parent nodes if a cost metric indicates that this will result in a higher quality tree. Kopta et al. adapt the notion of tree rotations for use in interactive updates [KIS*12]. As with other update methods, a refit of the tree is first performed. During refitting, this update method has the option to exchange child nodes between two parents if it is expected to result in a lower SAH cost for the subtree. Once the exchange occurs, refitting for those nodes is performed again. In addition to this, splitting and merging of leaf nodes is allowed. For example, if two triangles referenced by a leaf diverge considerably, it may make sense to split that node in two to reduce the traversal cost of the subtree. The paper investigates these techniques on both CPU and GPU platforms, and demonstrates competitive time to image figures compared to simple refitting and full recursive binned SAH rebuilds.

Although BVH updating can be useful, they do not represent a replacement for full rebuilds, as they are applicable in only limited circumstances. Furthermore, for many applications such as photon mapping, refitting is not applicable as insufficient geometry coherence exists over the course of rendering. As such, they represent a way to augment rebuilding approaches, as opposed to being a complete, alternative method.

2.5 Special Purpose Ray-Tracing Hardware

A number of micro-architectures for ray-based illumination algorithms have been proposed. This section begins by first looking at architectures for ray-tracing which have been developed in academic settings and reported in the academic literature. These architectures are categorised as either *fixed-function*, *fully programmable* or *hybrid fixed-function/programmable* designs. Secondly, a summary of commercial products utilising ray-tracing hardware is given. Finally, a brief description of the related class of volume rendering architectures is presented.

2.5.1 Fixed-Function Designs

Kobayashi et al. [KSS*01] present an architecture for an application-specific integrated circuit (ASIC) design for producing photo-realistic images known as the 3DCGiRAM. The proposed architecture supports two modes of operation; radiosity mode for calculating view-independent globally diffused intensities and a ray-tracing mode for calculating view dependent intensity calculations. The processor is first run in radiosity mode once to calculate the view independent effects, and then the ray-tracing mode is activated repeatedly to render subsequent frames. The unit itself consists of a *3D Differential Analyser unit* to traverse the uniform grid structure utilised by the architecture. In addition, it features thirty two *intersection calculation units* (ICUs), a *secondary ray generation unit* and an *intensity calculation unit*. An *object space table* (OST) maintains a mapping of grid elements to their corresponding memory locations. Furthermore, an *object existence table* (OET) keeps track of which grid cells are empty, thus preventing unnecessary accesses to the OST, reducing latency. Traversal is performed with the hardware 3DDA, and using the OST and OET, triangles located in the visited voxels are fetched from a geometry memory. Rays are intersected with triangles in the ICUs, and partial results are stored in a frame buffer cache. Depending on the surface, secondary rays are generated by the secondary ray generation unit. Results show that the 3DCGiRAM is capable of sustaining one ray/triangle intersection per cycle at peak performance.

Todman and Luk [TL01] discuss the use of reconfigurable hardware for accelerating the ray-tracing process. They focus on the intersection calculation (a ray/sphere intersection) as it represents one of the most expensive operations of the process. Although little detail is given of the actual hardware design, the authors do note the use of a breadth-first approach to ray-tracing. That is, instead of tracing a primary ray and then immediately tracing all secondary rays related to that primary ray, all primary rays are processed in one batch, the results are stored, and then all secondary rays are traced in one batch. The purpose of this is to optimise the number of bus transactions, and increase the efficiency of the pipeline.

Schmittler et al. [SWS02] present an architecture for real-time ray-tracing of static scenes on a single chip. The *SaarCOR* chip consists of three main units: the *ray generation and shading unit* (RGS), the *ray-tracing core* (RTC) and a *memory management unit* (RTC-MI). The system supports multiple RGS and RTC functional units in cascade. A master unit assigns pixel coordinates to each RTC, which carries out all computation for that pixel. The RTC traverses rays through a BSP tree of the scene. A *list unit* fetches the required triangles when a leaf node is hit, and an intersection unit intersects rays with triangles. The design utilises packet-tracing techniques [WSBW01] to amortise operations over groups of rays. The traversal of a packet is split over each RTC, with each performing computation relating to an individual ray. The results of each RTC are then combined to produce the final result. In this manner, n RTC units in cascade appear as one larger RTC unit. Multiple packets are executed simultaneously, via a multi-threading approach, which mitigates the impact of memory stalls. Caches are maintained for each

of the traversal, list and intersection units.

The SaarCOR architecture was further developed to accommodate dynamic scenes and to demonstrate a field-programmable gate array (FPGA) prototype [SWW*04]. The approach to dynamics taken is similar to that of Wald et al. [WBS03], which involves splitting the scene into objects, maintaining a local kd-tree for each object, and a top-level kd-tree over the objects. In this manner, only the top-level kd-tree has to be rebuilt when objects move throughout the scene, provided that the objects are rigid bodies. A major addition to the architecture is the *transformation unit*, which is responsible for transforming rays into the local coordinate spaces of the objects. As this transformation unit involves a matrix multiplication, it translates into relatively expensive hardware real estate. This motivated the reuse of the unit for other purposes throughout the design. The transformation unit is reused by the intersection algorithm and the generation of rays is reformulated as a matrix transformation to further increase unit utilisation. The list unit is also updated to include mailboxing to eliminate redundant intersection calculations. The prototype itself consists of four traversal units operating in parallel. However, the limited capacity of the FPGA results in a single intersection unit on chip. Thus, packets must be intersected sequentially. To allow for larger scenes, the architecture utilises a virtual memory scheme [SLS03] to stream data from disk as needed. Although very positive results are reported, memory bandwidth, especially for large scenes, is identified as the major bottleneck of the architecture.

Fender and Rose [FR03] propose a hardware ray-tracing solution utilising two FPGA chips. One FPGA is responsible for calculating ray object intersections, and is based upon the efficient barycentric coordinate test as described by Möller and Trumbore [MT97]. The intersection unit is pipelined and can only intersect a single triangle at a time, but does so in groups of three rays per triangle. However, the rays are independent and do not represent a packet. A register file is used to store the closest intersection points discovered for each ray. Another FPGA implements the traversal unit for the BVH utilised by the design. An interconnect exists between the two to allow for inter-chip communication. This hardware uses fixed-point arithmetic as opposed to floating-point to reduce hardware cost.

Park et al. [PhNP*08] present an architecture for ray-tracing on an FPGA chip. Similar to many other designs, the system consists of a *ray generation unit*, a *traversal and intersection unit*, and a *shading unit*. A kd-tree data structure is built on a host CPU, and sent over a USB interface to DRAM chips local to the FPGA board. Scene data is also stored in 512Mb of local DRAM on the FPGA board. The traversal unit differs from other designs in that it does not trace ray packets, but rather implements a full MIMD unit that traces individual rays. The hardware utilises caches for kd-tree nodes, primitives and textures to reduce access time to RAM.

CREMA is a fixed intersection architecture which proposes to utilise a *nano processor* for each triangle in the scene [OS07]. Each nano processor computes the intersection of a ray with its assigned triangle in parallel. A *selector* unit is used to perform a log reduction of all of the intersections. Although it is likely not feasible to store the entire scene on-chip, it is possible this design could be utilised in some way in a ray-tracing processor for subsets of the scene.

A recent traversal and intersection architecture known as the T&I engine is presented by Nah et al. [NPP*11]. It is a MIMD style processor which operates on single rays (i.e. rather than packets). The architecture consists of a number of units, notably a *ray dispatcher* (RD), twenty-four *traversal units* (TRV), six *list units* (LIST) and two varieties of *intersection unit*, eight of the first type and one of the second (IST1 and IST2). Separate caches are maintained for each of these, excluding the IST2 unit. The T&I core is designed to work with a kd-tree data structure using an *ordered depth first layout* (ODFL) [NPK*10]. Ray generation is assumed to occur in separate programmable shaders, and the ray dispatcher

is responsible for transferring such rays to the T&I core. Once in the core, the TRV performs a traversal step with the required kd-tree node utilising a short stack approach. If a TRV encounters a leaf node, it is passed to the LIST which is responsible for finding the primitives associated with the leaf. These primitives are then sent to the IST1, which perform a ray/plane test and a barycentric coordinate test to decide if an intersection is possible. If both of these tests have passed, the intermediate values are used to calculate the final intersection point in the IST2. The advantage of this is that data need not be fetched for the final intersection calculation if the IST1 can guarantee it will produce a negative result. The T&I engine utilises a latency hiding strategy on cache misses. If a cache miss occurs, a ray can be placed in a *ray accumulation unit* (RA), which postpones the ray's execution. Rays which require the same data are placed together in the RA unit. While fetching the data from main DRAM, processing can switch to other rays for which data is available. Once the data request can be fulfilled, stalled rays in the RA are preferentially output to the pipeline. Assuming a 500MHz clock, four T&I engines together demonstrate 44-1188 Mrays/s, which exceeds the performance of a GTX 480 GPU by around 5 - 10x.

2.5.2 Fully Programmable Designs

Aside from fixed function designs, programmable ray-tracing architectures have been demonstrated. A very early example of one such architecture is the TigerSHARK architecture [Hum96]. Rather than focusing on a single-chip solution like most other ray-tracing hardware, this architecture is proposed for inclusion in a large scale cluster-type environment. The basic atom of this architecture is a processing element containing a group of fully-featured programmable DSP chips which are connected to a central memory used for storing the scene database. Each chip is responsible for performing ray/primitive intersections. The chips operate in lock-step against the memory, meaning that they all must read the same word at every access. Each processor has a small amount of local memory and the processors are connected via serial communication ports. Rays are distributed to each DSP chip along the serial ports and they each intersect a different ray with the current primitive read from memory. Rays are distributed to the processors as coherently as possible, and a bounding volume subdivision is used. These processors are implemented on PCI cards, and multiple PCI cards may be placed in a host system, with the object database distributed across the cards and the same rays supplied to each. Results are collected from the processors by a host and closest intersections resolved. For additional parallelism, multiple hosts may be used and an image space partitioning may be applied. The authors compare their system to the ART AR250 chip (discussed later in Section 2.5.4) and conclude that the primary advantages of their system lie in flexibility and also price/performance ratio.

Another fully programmable design is the Ray Processing Unit (RPU) [WSS05]. The design of the RPU is quite influenced by the architecture of conventional GPUs. Each RPU consists of multiple programmable *Shader Processing Units* (SPUs), which include their own vector instruction set. Each SPU is multi-threaded and avoids memory latencies by switching threads when necessary. The SPUs can be used for a variety of purposes, including geometry intersection and shading determinations. SPUs are grouped into *chunks* containing a small number of units. Each SPU in a chunk operates in a lock step manner with the other SPUs in its chunk. Multiple asynchronous chunks work in parallel to complete a task. Coupled with each SPU is a dedicated *Traversal Processing Unit* (TPU), which utilises a kd-tree data structure, and can be controlled by the SPUs via the instruction set.

The TRaX architecture [SBK*08, SKKB09] is a fully-programmable ray-tracing architecture which consists of a number of top-level parallel cores connected to memory via a shared L2 cache and also

connected to a frame buffer memory. Inside each top-level core is a set of *thread processors*. Each thread processor possesses its own private functional units, and all thread processors in a core share a group of larger, less frequently used functional units to maximise utilisation. The thread processors contain state information for a number of threads, and each thread corresponds to a ray. The design is optimised for single ray execution and therefore does not rely on coherent packets. Results indicate that performance compares favourably to the RPU [WSS05], and allows for more flexibility in programming. A very detailed exploration of the design space for a very similar architecture has also been published [KSBD10], with additional data illustrating not only high performance but also greater performance per die area than competing GPU implementations. A mobile processor utilising a very similar architecture has also been presented [SKBD12].

A further evolution of TRaX is seen in the STRaTA architecture [KSS*13]. STRaTA follows the same basic TRaX design but includes a number of important additions which are designed to reduce bandwidth and power consumption. The first addition is that the spatial index (a BVH in this case) is explicitly decomposed into *treelets* [AK10], small subtrees of the BVH that fit entirely into the L1 cache. The goal of using treelets is to improve L1 cache efficiency by scheduling rays which traverse the same treelet in close succession. To achieve this, a portion of the L2 cache is replaced with dedicated ray queues, and rays which cross the boundary of the current treelet are collected in these queues for later processing of their next associated treelet. The second major improvement over TRaX is the inclusion of reconfigurable pipelines. These pipelines allow execution units to be dynamically connected into a pipeline which implements a given important ray-tracing operation (such as a ray/box test). Energy per operation is reduced as multiple instructions need not be fetched for each data item which must undergo the operation, as items are simply passed into a pre-configured pipeline. The reconfigurable pipelines are under software control, allowing for a degree of flexibility. Combining both techniques in the STRaTA processor yields significantly better energy efficiency compared to TRaX, while not significantly impacting rendering performance.

Govindaraju et al. [GDS*08] conduct an investigation of a novel hardware/software co-design for ray-tracing. Insights from profiling a novel software ray-tracing architecture, named *Razor*, are used to inform the hardware design process. The hardware architecture consists of a set of tiles, each containing a number of programmable cores which share an L2 cache which is private to that tile and connected to each core via a crossbar network. Each core possesses L1 instruction and data caches. The cores utilise a specialised instruction set architecture (ISA) appropriate for ray-tracing. Each core allows simultaneous multi-threading with between 2 - 8 threads and 4-wide SIMD units are employed within cores. Each tile is designed to accommodate an *accelerator block*, which may contain specialised hardware for graphics operations. However, the authors do not implement the accelerator block for this particular publication. The software application maps to the hardware by distributing chunks (sections) of the image plane to individual tiles on the processor, and within each tile, packet tracing is performed by the cores. The authors find through analysis that individual chunks in the image require relatively disjoint subtrees in the data structure, and therefore each tile builds its own copy of the data structure on demand according to which nodes in the data structure it needs to complete its set of rays. This allows parallelism and also removes the need for synchronisation. A system simulator, in combination with an analytical model for larger experiments, is used to determine that with 128 cores running at 4 GHz, the system is predicted to be capable of sustaining 10 - 75 Mrays/second.

The StreamRay architecture [RGD09] utilises the stream filtering concept [GR08], where a working set of rays are partitioned into active and inactive groups by a filter depending on their traversal or

intersection branch decisions. This extracts high ray coherence as it gathers all rays following the same path in the group before beginning the next processing step. The hardware itself consists of two main units, the *ray engine* and the *filter engine*. The ray engine consists of a parallel address fetch unit (AFU), a control block and a scratchpad memory. Also included in this unit are a set of paired, multi-banked buffers. The AFUs assemble streams into one of the buffers in each pair, while the filtering engine operates on the stream in the other pair. Thus the two operations can occur in parallel. The filtering engine consists of a number of programmable SIMD cores operating in parallel, which implement the filtering for each step of the ray-tracing process. An on-chip interconnect allows data transfers to occur between the ray and filter engines, as well as between neighbouring SIMD cores within the filtering engine. Results based on a 500MHz implementation of the design indicate that high SIMD efficiency and frame rates can be achieved while running a Monte Carlo ray-tracer on the system.

Aila and Karras [AK10] present a study of a hypothetical programmable architecture designed to effectively handle incoherent ray distributions. Their design is based on recent commodity GPUs (Fermi-type architecture) and consists of 16 processors, each featuring 32-wide SIMD lanes and 32x32 threads. A private L1 cache is included in each processor, as well as a shared L2 cache between processors. Each processor also includes a *Launcher*, which schedules rays to the SIMD units. This architecture focuses mostly on reducing memory bandwidth requirements, as it assumes this to be the primary bottleneck for future workloads. Work compaction is used to improve SIMD utilisation and a stack caching mechanism similar to a short stack is used. The authors also introduce the concept of *treelets*, which is fundamental to their scheduling mechanism. They first partition the spatial index into subtrees which are small enough to fit either in the L1 or L2 cache, and create a ray queue in memory for each treelet. During traversal, rays that enter a region of the tree corresponding to the treelet are collected in a queue. Queues are then chosen by the scheduler for processing, and since all rays contained in a queue require the same treelet data, good cache performance can be achieved, lowering external memory bandwidth. The authors compare two alternative schemes for deciding which queues to schedule next, which they call *lazy* and *balanced* scheduling, as well as comparing a number of ways for allocating these queues in memory. The authors show that their performance is insensitive to the order in which rays are supplied to the unit, which they argue improves the usability of the system. Overall, the design is shown to reduce total memory bandwidth requirements for ray traversal and intersection by up to 90% compared to a baseline system.

2.5.3 Hybrid Fixed-Function/Programmable Designs

Sanchez-Elez et al. [SEDT*03] present a mapping of ray-tracing to a hybrid reconfigurable hardware processor. The architecture is based on the MorphoSys reconfigurable processor which includes an 8×8 array of *Reconfigurable Cells* (RC array), a TinyRISC instruction set processor [ACG*92], a frame buffer and a cache. For the ray-tracing architecture, additional specialised units are added, including a *pointer buffer*, a *pointer update unit* and a *Spatial Partitioning Tree* (SPT) buffer. The pointer update unit provides hardware support to offload pointer calculations for traversal from the TinyRISC processor as it is deemed too expensive. The pointer buffer is used to offload stack operations. The SPT buffer controls access to main memory, and includes eight banks, one for each column of RCs. The SPT can detect duplicate requests from columns of RCs and combine them to reduce bandwidth requirements. The columns in the RC array are capable of operating in a 8-wide SIMD fashion, and are used to implement the majority of the algorithm, including traversal and intersection of rays. A context memory supplies the configuration data specifying the current kernel implemented in each RC. An octree data structure is

used as a spatial index and a SIMD algorithm for its traversal is proposed. The implementation supports primary, shadow and reflection rays and also sphere mapping and spherical harmonic coefficients to model environment light sources. The chip operates at 300MHz, with a 32MB external memory operating at 133MHz and consumes only 1W. The architecture exhibits competitive performance in total rays per second compared to a variety of other software and hardware ray-tracing implementations for small scenes. Earlier work on implementing ray-tracing on this architecture (using block fixed-point arithmetic) was also presented by mostly the same authors [ATD*02].

The DynRT architecture [WMS06] is designed to provide limited self-contained support for dynamic scenes. It achieves this by employing its novel B-KD tree data structure, that is updated by an *Update Processor* over subsequent frames. In addition, a *Skinning Unit* implements a skinning model over the geometry for updating of the scene. The design also features dedicated traversal and intersection hardware which traces packets of rays in a multi-threaded scheme to hide memory latencies. However, the design relies on CPU support for the initial construction of the B-KD tree and through refitting can efficiently support coherent dynamic scenes. A further publication by the same author extrapolates a similar design from an FPGA to an ASIC, and estimates the memory bandwidth and die area requirements, as well as the overall performance of such an approach [WBS06]. This project was also covered in detail in Woop's PhD thesis [Woo06].

Steffen and Zambreno present a hybrid architecture for tree traversal in two publications [SZ09, SZ10]. The idea is based on a traditional GPU architecture and proposes to incorporate a fixed-function unit for traversal of the upper levels of the hierarchy and to use conventional stream processors for the lower traversal steps. The fixed function component of the traversal operates on a novel hash function based concept. To begin the process of traversal, a uniform grid is first built over the scene. A kd-tree data structure is then built over this in order to group cells together in the uniform grid, such that the number of leaves in the kd-tree is normally much fewer than the number of original grid cells. Each of these cell groupings is referred to as a *GrUG group*. Once a set of GrUG groups is determined, a custom data structure of any type may be built inside each group, forming a hybrid data structure. During traversal, a hash function first takes a ray and computes the cell of the original uniform grid that it originates in. This computed cell is then mapped to the GrUG group in which it resides. This GrUG group is then mapped to the root of the custom data structure within the GrUG and traversal can continue with programmable processors. This scheme allows much of the traversal operation to be performed with very few operations and no external memory reads. Compared to a conventional kd-tree data structure, adding the GrUG scheme is shown to improve traversal performance by an average of 1.6x and can also reduce bandwidth per frame by a similar factor, while partially preserving programmability of the traversal operation.

The *Mobile Ray-Tracing Processor (MRTP)* [KKK10, KKK12] is a programmable design which takes a unique hardware approach to solving SIMT/SIMD utilisation problems due to diverging ray distributions. The design is targeted particularly for mobile platforms. The basic architecture consists of three *reconfigurable stream multiprocessors (RSMP)* which are used to execute one of three kernels: ray traversal, ray intersection and shading. These kernels are stored in three on-chip instruction memories which are shared by the RSMPs. Kernels can be adaptively reassigned to RSMPs to enable load balancing. Rays are passed between RSMPs via a crossbar network to complete all phases of ray-tracing. Each RSMP is a SIMT processor consisting of 12 *Scalar Processing Elements (SPE)*. The unique aspect of this architecture arises from the ability of the 12 SPEs to reconfigure into either a 12-wide regular scalar SIMT operation, or a 4-wide 3-vector SIMT operation. The system uses the regular scalar SIMT mode for traversal and shading, and reconfigures into the vector mode for triangle intersection, which exhibits

greater code divergence. The goal of this organisation is to improve datapath utilisation by relying less on ray coherence while executing typically divergent code such as ray traversal. Similar to the TRaX architecture, special functions such as square root and reciprocal which are expensive in terms of area, but less frequently used, are shared among SPEs. While mostly programmable, the architecture employs a small amount of fixed-function logic in the form of a ray generator. The authors report that performance of this design compares favourably to the DRPU [WBS06], while preserving more flexibility due to the use of programmable components for a greater portion of the pipeline.

Davidovic et al. [DMS11] present a traversal and intersection architecture based on the RPU, known as the *ray traversal engine (RTE)*. It is proposed that this unit be incorporated into a programmable GPU. In this work, the authors consider a range of configurations for their design, but the design consists basically of an *I/O unit*, a *Traversal Processor* and a *Geometry Processor*. The I/O unit is responsible for reading and writing ray queues to/from memory. The traversal processor is responsible for traversing either a B-KD tree or a BVH, depending on the configuration. The geometry unit is responsible for both ray/primitive intersection and also ray transformation, as two-level hierarchies are supported. Each unit is connected to main memory via a separate L1 cache, and again depending on the configuration, a L2 cache may be included, which is shown to improve performance. The unit processes threads containing four rays each, and supports different threads in each pipeline stage, with no switching overhead. This design can also be configured to schedule rays based on the treelet concept [AK10] which splits the data structure into sub-hierarchies which can be fit entirely inside the cache. The authors also consider the effect of both continuation-type shading and also tail recursive shading on their design's performance. Results indicate that this design can operate at clock frequencies as high as 2 GHz, and deliver high ray throughput (around 100Mrays/sec) with low die space and bandwidth requirements.

The *Samsung Reconfigurable GPU based on Ray-Tracing* (SGRT) [LLN*12, LSL*13] is a mobile GPU designed for ray-tracing applications. The primary building block of this GPU is the SGRT core, which consists of two principal components. The first component is a fixed-function *T&I core* which is responsible for traversal and intersection of rays. The T&I unit of the SGRT is an updated version of that presented by Nah et al. [NPP*11]. It is adapted to operate on a BVH data structure (as opposed to a kd-tree), and utilises a novel parallel-pipelined structure which reduces average ray latency and improves cache performance [KLLH12]. The second major component of the SGRT is a programmable *Samsung Reconfigurable Processor* (SRP), which consists of a combination of a very long instruction word (VLIW) processor and a *coarse-grained reconfigurable array* (CGRA), and is designated for ray generation and shading operations. The CGRA consists of a number of inter-connected functional units and connected register files, that can be configured to perform a variety of operations. The SRP supports a standard C-type language, and its associated compiler extracts compute-intensive portions of the algorithm and targets these to the CGRA array, with the remaining portions (such as control flow) mapped to the VLIW processor. The top-level of the GPU consists of a number of SGRT cores coupled with a multi-core ARM processor which is used for BVH construction. Cycle-accurate simulations and FPGA verification indicate that the design offers ray-tracing performance on-par with recent GPU implementations, while achieving a much higher performance-per-area, indicating that this GPU has very good potential for achieving high-performance ray-tracing in a mobile environment.

2.5.4 Commercial Products

A small number of commercial products featuring hardware ray-tracing support have been released.

The AR series of processors (AR250, AR350) were a commercial hardware solution for high performance ray-tracing, produced by ART VPS [Hal01]. A number of products utilising these processors were available, including the PURE system, which was a PCI card containing eight such processors, and also a RenderDrive machine which included 48 AR processors. The processors themselves consisted of fixed geometry intersection and ray generation hardware, as well as on-chip data caches and a programmable shading processor. Each processor contained multiple cores and several processors could cooperate to produce an image through ray-parallel computation and broadcasting of the scene database. The system conformed to a standard RenderMan-type interface, which allowed integration with existing systems. As the AR processors were part of a commercial venture, only limited technical information about them is available. However, unlike other hybrid systems such as the RPU or SGRT, it appears that specific hardware support for ray traversal was not included in the design. A publication dealing with the extension of the AR processor's capabilities to include support for path-tracing via special RenderMan shaders was also published [CRR04].

A more recent commercial venture producing custom hardware for ray-tracing is Caustic Graphics [Ima12]. In 2009, Caustic announced the CausticOne accelerator board which reportedly contained hardware support for handling incoherent ray distributions with great efficiency. The card contained two FPGAs, presumably for implementing the ray-tracing logic. The CausticOne card was designed to work in cooperation with a CPU or GPU for shading and other calculations. In 2010, Caustic was acquired by Imagination Technologies [Ima10], which manufactures the PowerVR line of graphics accelerator IP. A recent SIGGRAPH presentation has revealed some technical details of this technology [KKW*13]. The presentation shows the use of fixed-function ray/AABB intersection pipelines for BVH traversal, coupled with programmable shader hardware. In addition, details on memory bandwidth optimisation via queue-based ray coherence extraction have been shown. Numerous patents relating to this technology have also been filed, including hardware and methods for novel bounding volume techniques [PMSP12], parallel intersection and shading [PMSP12] and management of ray groupings [PSGC10].

2.5.5 Volume Rendering Architectures

A related class of hardware architectures are those designed for volume rendering. Volume renderers are used extensively in medical and scientific visualisation [DCH88, SG93, SPLB97, BHWB07]. Volume rendering shares the concept of ray shooting and traversal with other ray-based algorithms, but it differs significantly from the algorithms previously described in this chapter on many details. As volume rendering is not quite the focus of this thesis, the author will restrict discussion of these architectures to a few examples for the sake of brevity.

The VIRIM system [GPR*94] is an architecture designed for real-time volume rendering. The design consists of a number of modules which are connected to a host system over a VME bus. Each module consists of a dedicated rotation board, and a DSP board. Four such modules are required for real-time performance. The rotation board includes a rotator unit comprised of a *geometry processor*, an *interpolation processor*, and a *gradient processor*, and is used for rotating voxels to arbitrary viewpoints. The DSP board contains 16 programmable DSP chips running at 40MHz which can be used to implement a variety of ray-casting algorithms. The host system executes software to provide a graphical interface and handle communication for user input. This system is targeted towards medical use and various applications in this field are discussed.

The VIZARD system [KS97] is a microarchitecture targeted for interactive volume rendering. The

system is based on a PCI accelerator card. Two such cards are included in a host PC and operate in parallel. The PCI cards themselves are implemented largely with several reconfigurable hardware components (Xilinx FPGAs). Two FPGAs on each card are designated to ray-casting operations. A custom cache controller based on the Manhattan distance of voxels relative to each other is implemented in another FPGA, and is connected to a 64Kx32 SRAM which stores the data. A multiply-accumulate unit is also present on the board as a separate chip, as well as a PCI interface implemented with another FPGA. The entire scene database is assumed to fit entirely into the host processor's main memory. Before rendering begins, various pre-processes including segmentation, precomputed shading and dataset compression are performed in software. When rendering begins, load balancing is achieved by assigning screen tiles to each of the four ray-casting engines present on the two boards. An updated version of this system, the VIZARD II, is presented in a second publication [MKS98]. The VIZARD II system includes dedicated memory on each of the PCI cards to hold the dataset, the data compression is removed, and precomputed shading is no longer utilised such that interactive changing of shaders during rendering is possible.

The VolumePro Realtime Ray Casting system [PHK*99] is a single chip solution for fast volume rendering. It is designed to achieve real-time performance with datasets of up to 256^3 voxels. The VolumePro system consists of a PCI accelerator board which includes a vg500 processor for ray-casting and a array of SDRAM DIMMs for storing voxel data. The vg500 chip operates at a clock frequency of 125MHz. The internal architecture of the vg500 chip consists of four rendering pipelines, each containing hardware for interpolation, gradient estimation, classification and shading, and compositing. 2MBits of on-chip SRAM is also included. Four 16-bit interfaces to off-chip SDRAM voxel memory are present. Four SDRAM DIMMS may be connected per interface, and using 64MBits DIMMs, as much as 128MB of on-board SDRAM can be included. The authors also discuss the possibility of connecting multiple PCI cards, or including multiple vg500 processors and SDRAM banks on a single board for greater performance.

Summary

In this chapter, the widespread use of the ray paradigm of rendering was illustrated with a look at a variety of ray-based illumination algorithms, ranging from those used in interactive renderers to those more suitable for offline rendering tasks. The concept of the spatial index was introduced, and an in-depth look at the BVH was given. The wide variety of high performance BVH construction algorithms was detailed, highlighting the importance of efficient construction of this spatial index for interactive ray-tracing systems. Furthermore, a detailed state-of-the-art on specialised microarchitectures for ray-tracing was presented.

In the next chapter, the author will build upon some of the existing research discussed here, and present the first design for a specialised microarchitecture for BVH construction for ray-tracing applications.

Chapter 3

Design and Implementation

THE primary contribution of this thesis is a novel microarchitecture for fast and efficient construction of bounding volume hierarchies for ray-tracing. In this chapter, the design and implementation of the system is documented in detail. This chapter begins by first identifying a number of guiding design decisions which were adopted early in the design process. Next, an overview of the final design is given, with a high-level description of its operation. Each major design unit is then treated in turn, with detailed schematics presented for each. Finally, the author's implementation is described, including information on the evaluation and testing framework put in place for the project.

3.1 Broad Design Decisions

To bring any complex hardware design to fruition, a number of design decisions naturally must be made. In the following section, the reasoning behind the major design choices for the BVH construction hardware is described.

- **Fixed-Function Hardware** The first major design decision to consider is what type of processor is to be built. The considerable variety of ray-tracing hardware discussed in Chapter 1 illustrates that many approaches are of course possible, from flexible programmable solutions utilising specialised instruction sets, to ultra-efficient fixed-function devices. Early in the BVH builder design process, it was decided to proceed with a fixed-function design for a number of reasons.

  Firstly, fixed-function designs naturally lead to optimally efficient hardware for a given computation, as they can be completely streamlined for a given task. BVH construction is a *repetitive* and *high utilisation* task in ray-tracing, justifying the expense of including fixed-function support for it in a graphics processor. Furthermore, many researchers have predicted that in future processors, die area will be plentiful, with power being the limiting factor [Tay12, HFFA11]. This situation mitigates the cost of including special-purpose logic on the die, if such logic is powered off when not in use.

  Secondly, the inherent loss of flexibility of a fixed-function design can be justified for a spatial index construction solution, especially if integrated into a dedicated ray tracing chip. Spatial index structures allow for accelerating the search for ray/primitive intersections, for which there exists a single valid result. Regardless of the type of spatial index used, the result of a ray query will be the same. If the spatial index is fixed in hardware, any ray-based illumination algorithms may

still be implemented without major restrictions, while benefiting from a highly efficient hardware accelerator.

Finally, a number of microarchitectures incorporating fixed-function traversal of spatial index structures have been proposed [Woo06, NPP*11, LSL*13]. In this case, fixed-function construction of the spatial index is a natural counterpart to these architectures, and would have limited additional impact on system flexibility.

- **Binned SAH BVH Construction** If it is decided to pursue a fixed-function approach to spatial index construction for ray-tracing, then it is important to carefully select an efficient and useful spatial index structure. The first decision to be made is which spatial index structure type to construct. As shown in Chapter 2, the BVH has become by far the most widely used structure in interactive ray-based illumination algorithms. This has occurred for a number of reasons; BVHs are typically faster to construct compared to alternative structures such as kd-trees, many updating schemes are available for BVHs, and memory footprint is generally predictable. Based on these observations, it was reasoned to be the most useful structure for modern interactive ray-tracing systems, and thus the BVH was chosen for hardware implementation in this project. Furthermore, heterogeneous ray-tracing processors utilising fixed-function traversal of BVHs already exist, and could directly benefit from hardware construction of the BVH [LSL*13].

  The second major decision to be made in regards to the type of data structure to be supported relates to the construction strategy which is to be adopted. As shown in Chapter 2, a considerable array of BVH construction strategies is to be found in the present literature. At the beginning of the project, two major BVH construction strategies were in use in interactive ray-tracing systems: top-down recursive builders and LBVH/HLBVH type builders. It was eventually decided to proceed with a binned SAH top-down BVH builder for a number of reasons. Firstly, binned SAH methods generally produce higher quality hierarchies compared to LBVH methods. Secondly, many hybrid LBVH builders utilise binned SAH methods for part of their hierarchy, in order to preserve reasonable tree quality [LGS*09, GPM11]. The potential for the binned SAH method to be useful in both types of builder greatly influenced the decision to proceed with this construction strategy.

- **IEEE 754 Floating-Point Compliance** Most commodity computer hardware supports floating-point arithmetic through floating-point ALUs in the datapath. Interestingly, some proposals for ray-tracing hardware, and indeed, software, have advocated the possibility of using fixed-point or integer arithmetic for many parts of the pipeline [HK07]. This is especially interesting from a hardware perspective, as it has potentially significant implications for circuit area and power consumption. On the other hand, difficulties arise when utilising fixed-point arithmetic, which can even produce rendering artifacts in the resulting image. In order to ensure a good degree of numerical precision, and to facilitate easy integration with existing systems, it was decided that the BVH builder would utilise IEEE 754 compliant floating-point arithmetic [IEE08] for all stages of the construction process. The investigation of fixed-point arithmetic is left as future work.

3.2 System Overview

With these design guidelines in mind, the basic architecture for the BVH construction hardware was arrived at. At a first approximation, the design is split into two major components, the *upper builder* and the *subtree builder*, shown in Figure 3.1. The upper builder directly interfaces to external memory and is capable of processing tree nodes encompassing an arbitrary number of primitives. It is thus used for processing larger nodes present in the upper levels of the hierarchy. The second major component is the *subtree builder*, which allows for construction of BVH nodes referencing fewer primitives than a fixed maximum threshold, determined by the capacity of the internal memory resources of the subtree builder. Both the upper and subtree builders are designed to operate on the AABBs of scene primitives. Also visible in Figure 3.1 is a DRAM interface consisting of a number of *RAM pairs*. Each RAM pair consists of two memory channels. These memory channels need not be physical memory channels, but could also be virtual ports, provided that sufficient bandwidth was provided to each port from the underlying memory system.

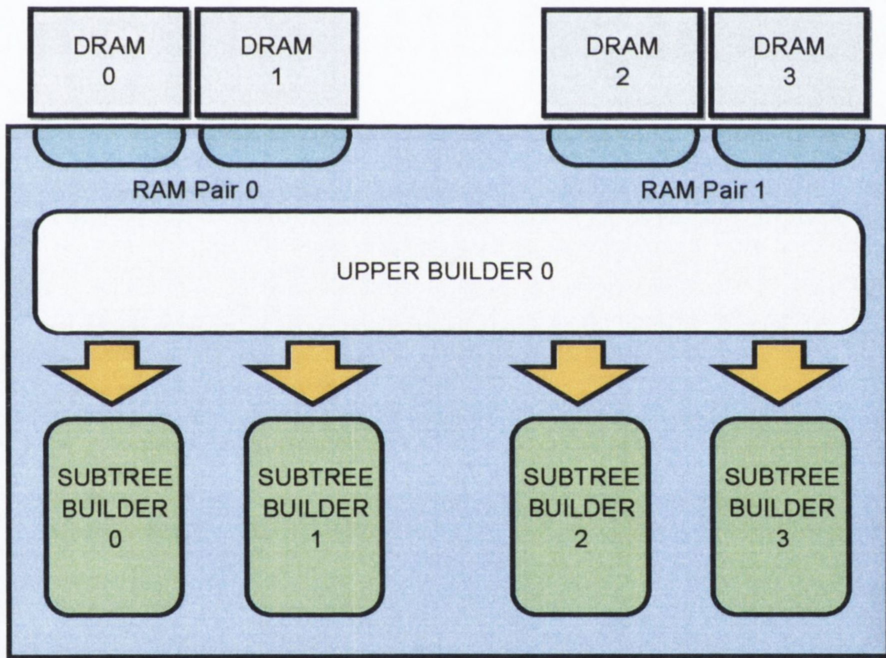


Figure 3.1: *Top-Level Overview of the BVH Builder Microarchitecture.*

Decomposition of the design into upper and subtree builders was adopted to most efficiently deal with the different properties of building the upper and lower parts of the hierarchy. Construction of the lower nodes in the hierarchy naturally exhibits very good data locality which allows for buffering small sections of the scene on-chip, and taking advantage of high-bandwidth, on-chip memory resources to exploit more parallelism. Construction of the upper levels of the hierarchy is typically limited by external memory, and thus a simpler solution is justified here. Having a clear distinction between these two units also improves system modularity and flexibility. It is envisioned that the hardware BVH builder should represent a piece of IP which will be useful to a variety of heterogeneous computing architectures. A modular design thus improves the ease of integrating the IP in other hardware contexts. In fact, examples of how subtree builders could be used by themselves in an alternative setup are described in Section 3.4.

Construction begins with the upper builder computing the root node of the hierarchy. The upper builder continues construction of the upper tree nodes until a node encompassing fewer primitives than a predetermined threshold is found. In practice, this threshold is envisioned to be of the order of several thousand primitives. This threshold is determined by the maximum capacity of the internal buffers of the subtree builders. The primitives corresponding to this node are then loaded from the upper builder into one of the subtree builders. The subtree builder then builds a complete subtree from these primitives. Once all primitives are passed to a subtree builder, the upper builder continues building the upper levels of the hierarchy, passing further subtrees to other subtree builders and stalling if none are available. The upper and subtree builders are therefore capable of operating in parallel.

The upper and subtree builders are largely the same hardware, except that the upper builder interacts with external DRAM, while the subtree builders interact with their own internal memory buffers. The core logic of the subtree builder is actually mostly a superset of the upper builder. A clear description of the system is readily achieved by first describing the subtree builder, and then explaining how it differs from the upper builder.

3.3 Subtree Builder Microarchitecture

The architecture of the subtree builder is shown in Figure 3.2. A relatively small instantiation is illustrated, so as to maintain clarity. The architecture is designed to operate on the AABBs of scene primitives, as is common with high-performance BVH builders, and is therefore suitable for any primitive type for which an AABB can be calculated.

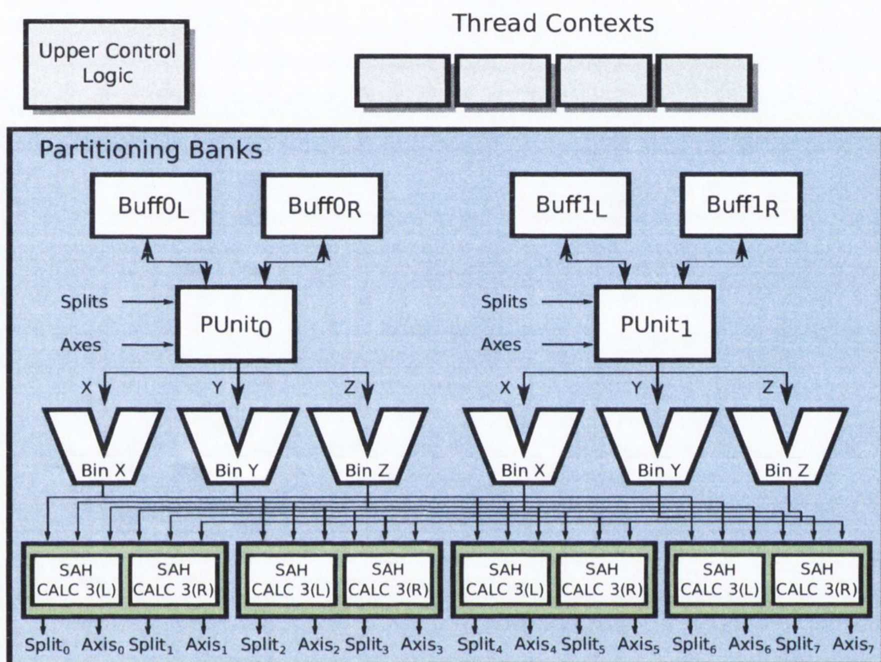


Figure 3.2: *Top Level-Overview of the Subtree Builder Microarchitecture.*

On a behavioural level, the subtree builder implements the classic binned SAH recursive algorithm for tree construction, similar to that presented by Wald [Wal07]. The subtree builder consists of a number of units which implement the various stages of this recursive algorithm.

The first units of interest are the *primitive buffers*. Primitive buffers are instantiated in pairs, and two pairs are visible in Figure 3.2, labelled *Buff0* and *Buff1*. Primitive buffers are a set of on-chip, high-bandwidth/low-latency buffers (similar to a cache memory). The purpose of the primitive buffers is to hold primitives on-chip as they are processed by the subtree builder.

Primitives are stored in the primitive buffers as 216-bit AABB-index pairs, with the index consuming 24 bits, with the remaining 192 bits consumed by the AABB. The index is used to map each AABB to the primitive it originated from, so primitives can be retrieved for intersection during traversal. This index-AABB pair format was arrived at by considering the storage requirements of a number of alternative formats. One such alternative is to store primitives (in most cases, triangles) directly in the primitive buffers. However, this option significantly increases the required memory resources in the subtree builder, as a 32-bit float-point triangle consumes 288 bits of storage. Furthermore, storing the triangle directly would either require recalculating or storing its AABB also. Moreover, operating on AABBs provides a simple way to handle a wide variety of primitive types, as any primitive for which an AABB can be calculated is supported. The data width of these buffers is set large enough for a full primitive AABB to be read in each cycle. They could also be implemented with several narrower memories in parallel.

Connected to each pair of primitive buffers is a *partitioning unit*. Two partitioning units are visible in Figure 3.2, labelled $PUnit_0$ and $PUnit_1$. The purpose of the partitioning units is, given an SAH split in a certain axis, to read a vector of primitives from the primitive buffers to which they are connected, and to partition those primitives into one of two new vectors, depending on which side of the splitting plane they reside. Each partitioning unit is hardwired to one pair of primitive buffers, and only that partitioning unit can access the primitives contained in that buffer pair. Primitive buffers are organised in pairs to allow partitioning units to swiftly partition lists of primitives, by reading primitives from one buffer and writing a newly sorted list into the opposite buffer concurrently. On subsequent partitioning operations on the sublists, the roles of the buffers are reversed, and the primitives are read from the buffer they were last written.

Below the partitioning units in Figure 3.2 is the logic which determines the SAH split for each node in the hierarchy. The subtree builder is capable of searching all three axes concurrently for SAH split candidates. SAH split determination is achieved in the subtree builder with two types of unit. The first such unit is the *binning unit*. Each partitioning unit is connected to three binning units, one for each axis (labelled Bin X/Y/Z). The purpose of each binning unit is to take a scene primitive AABB, to calculate its centre, and to determine in which SAH bin the centre lies in its designated axis. As will be explained in Section 3.3.1, partitioning and binning of primitives occurs in parallel, and so the binning units latch data from the output of primitive buffers as they are being read by the partitioning unit. The binning units must also keep track of the AABB of the hierarchy node currently being evaluated for splitting.

The binning unit passes SAH bin locations, and also forwards the original primitive AABBs to the last set of units, the *SAH calculators*. Eight SAH calculators are shown in Figure 3.2, labelled *SAH CALC*. Primitive AABBs and their chosen bin positions are fed into the SAH calculators which accumulate an AABB and a counter for each SAH bin in each axis. Once all primitives are accumulated, the SAH calculators evaluate the SAH cost for each possible split, and output the lowest-cost split found across all three axes. In the subtree builder, SAH splits are always calculated in pairs, which correspond to a pair of sibling nodes in the hierarchy. Multiple pairs of SAH calculators are used to support multithreading, which is described in Section 3.3.1.

The number of each type of unit in a given instantiation of the subtree builder can be altered somewhat independently. The following parameters control the instantiation of a subtree builder:

- ***Width*** (**w**) The width of the architecture. A width of w creates a subtree builder with w partitioning units, $w \times 3$ binning units and w pairs of primitives buffers. w is limited to being a power of two.

- ***Primitive Buffer Capacity*** (**Psize**) The number of primitives each primitive buffer can hold. $Psize$ can be any positive integer value. The number of primitives which can be stored in the subtree builder for processing is $Psize \times w$, whereas the total capacity of all primitive buffers is double this value.

- ***Number of Concurrent Threads*** (**Tsize**) The number of different construction paths in the tree that the subtree builder can pursue at any given time. The subtree builder utilises context switches in order to hide the latency of the various units in the design and to keep the pipelines as full as possible. One pair of SAH calculators is instantiated for each thread, yielding $Tsize \times 2$ SAH calculators. $Tsize$ must be a power of two. Multithreading in the subtree builder is described in detail in Section 3.3.1.

3.3.1 Sequence of Operations

Now that the function of each unit has been established, the sequence of operations that the subtree builder performs to generate a hierarchy can be described. Coordination of the hierarchy construction is largely managed by the partitioning units, as they perform most of the decision making in the builder. The partitioning units manage the data flow through the builder, providing the other units with the data they need to perform their respective tasks. In this way, the other units are chiefly data path components. The *Upper Control Logic* shown in the upper left corner of Figure 3.2 coordinates only simple sequencing operations, such as controlling the initial filling of the subtree builder with scene primitives, as well generating external signals such as "READY" and "DONE" signals.

Before construction begins in the subtree builder, it is necessary to fill the primitive buffers with the scene primitives for which a BVH is to be constructed. Each subtree builder provides one external port for each primitive buffer pair which allows for filling that pair with primitives. During filling operations, only one buffer in the pair is filled, with the other being left empty. Ideally, an equal number of primitives will be randomly assigned to each buffer pair, so as to facilitate load balancing across units. Once filling is complete, a subset of the scene will reside in each buffer pair. Each partitioning unit is now in control of the subset of scene primitives which reside in the primitive buffer pair it is hardwired to. The bounding AABB of all primitives is also loaded into a register at this point, and this constitutes the root node of the hierarchy.

Once filling is complete, construction can begin. Figure 3.3 illustrates the state machine implemented inside each partitioning unit as a flowchart. The reader may also find it useful to refer to Figure 3.2 during this description. The builder begins in the idle state *S0*, waiting to receive a GO signal to begin construction. Once that signal is received, the *initial split phase* (state *S1*) is run. Each partitioning unit reads all primitives resident in its primitive buffer pair, and passes these to the binning units it is connected to (as shown in Figure 3.2). As primitives pass through the binning units, the root node of the hierarchy is used to determine the SAH bin in which the primitive lies. As three binning units are connected to each partitioning unit, a bin is resolved for each axis in parallel. As bin locations are determined, the binning unit passes these to one of the SAH calculators. When a partitioning unit has finished reading all of its primitives from the buffer, it moves to state *S2* where it waits to receive the split for the root node.

As shown in Figure 3.2, each SAH calculator is connected to all binning units. As each partitioning unit controls only a subset of scene primitives, and each binning unit is connected to only one partitioning unit, the SAH calculator must collect all primitive bin decisions from each binning unit to determine the split for the node. Once the split has been determined, it is passed back up to the partitioning units. Specifically, the SAH calculator outputs the chosen SAH split bin, the chosen axis and, importantly, the AABBs and primitive counts of the two resulting child nodes. Detailed descriptions of how the binning units and SAH calculators operate are given in Section 3.3.3 and Section 3.3.4 respectively.

Once the split information is available, the *partition phase* beginning at state *S2* can proceed. As each primitive buffer pair is hardwired to a single partitioning unit, only that partitioning unit can access the primitives in that buffer pair. Once the split for the root node is ready, the partitioning units each leave state *S2* of Figure 3.3 and move to the partitioning phase of the algorithm in state *S3*. In this phase, the partitioning units read each primitive in the current node from one of their connected primitive buffers, and determine on which side of the chosen split the primitive resides. Once this has been determined, the partitioning unit writes the primitives into one of two sublists in the *opposite* buffer of its primitive buffer

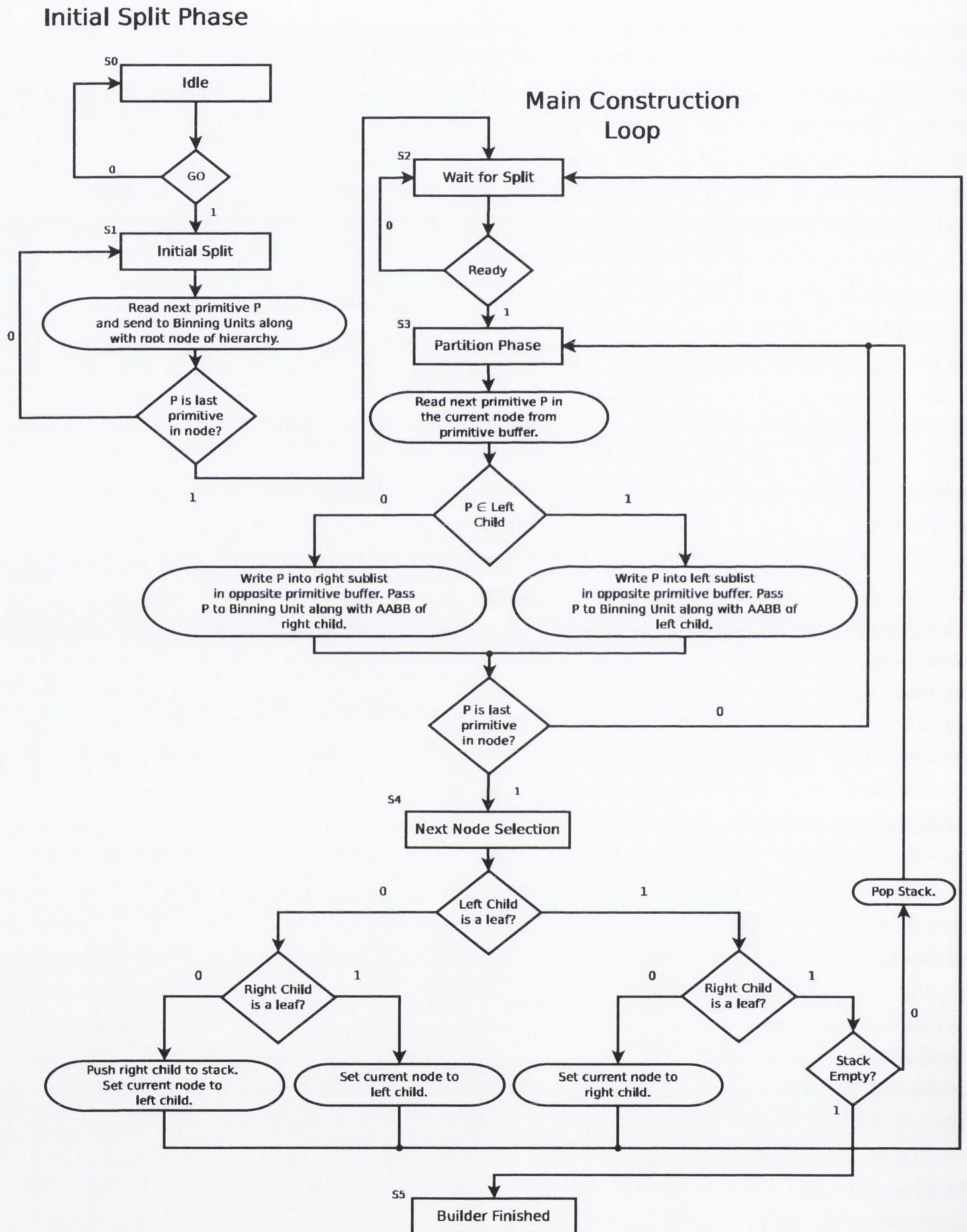


Figure 3.3: *Flow Chart of the Partitioning Unit operation.*

pair. Primitive buffers are arranged in pairs, as this allows for concurrently reading from one buffer and writing to another, which accelerates partitioning operations.

Partitioning is achieved in each pair by maintaining two address registers, a *lower* and an *upper* register, inside each partitioning unit. The lower and upper registers begin at the bottom and top address respectively of the subset of primitives that belong in the node currently being processed. These registers are then multiplexed onto the address of the primitive buffer as appropriate. In a real implementation, buffer pairs could be implemented with two physically distinct buffers, or alternatively a multi-port buffer. By performing this operation for each primitive in their primitive subsets, the partitioning units in effect cooperate to partition all primitives in a data parallel manner.

After a partitioning unit has placed each primitive into its correct sublist of primitives (left and right of the split), it moves onto the next phase (state *S4*), in which it decides how to recurse with hierarchy construction. Each partitioning unit will always make the same decision as to whether to proceed with the right or left child node (or to pop the stack). It is possible that, after a split, one or more child nodes will represent a leaf node. The current implementation of the subtree builder supports terminating construction at a specified maximum leaf size in terms of the number of primitives encompassed by the leaf. The logic for making this decision is visible in state *S4* of Figure 3.3. If neither child node is a leaf node, then the right child is placed on the stack, and the partitioning unit moves back to state *S2* to wait for the split for the left child to be calculated. If both child nodes are leaves, then the stack is popped to retrieve the next node to be processed. Since several partitioning units all partition a subset of the current node's primitives in their respective buffers, several partitioned lists exist, which when added together form the full list. To keep track of this information, a *wide stack* is employed. Wide stack elements include the full AABB of the pushed node, the chosen split position and axis for the node, and also separate primitive ranges for each primitive buffer pair detailing the range in which primitives reside in that particular pair for a given node. In the event of a stack pop, the partitioning unit can immediately proceed to state *S3*, as the required split information will be present on the stack.

The last piece of the puzzle is how the splits are determined for the two new child nodes, if indeed they are to be partitioned. In the subtree builder, determining the splits for the two new child nodes occurs in parallel with the partitioning of their parent node. The fact that partitioning is occurring means that the SAH split information (which includes the AABBs of the two resulting child nodes) is available. Therefore, all the information necessary to begin binning primitives into the new child nodes is available at the time of partitioning. During partitioning, primitives are not only written into the opposite buffer, but are also fed into the binning units at the same time. This fact is reflected in the decision logic of state *S3* in Figure 3.3. The binning units bin each primitive into either the left or right child, depending on which side of the partition it belongs to, by multiplexing the correct node AABB into its pipeline.

Since partitioning implies that two new nodes will be created, SAH calculators are placed in pairs (shown as green blocks at the base of Figure 3.2). The binning units output the bin decisions and primitive AABBs which are then fed into one of the SAH calculator pairs. If a primitive resides on the left side of the split in the previous node, it is fed into the left SAH calculator of the pair, otherwise the right. Both calculators in a pair operate concurrently. If both child nodes are to be split, the chosen split for the right child is placed onto the stack, along with its primitive ranges in each buffer pair.

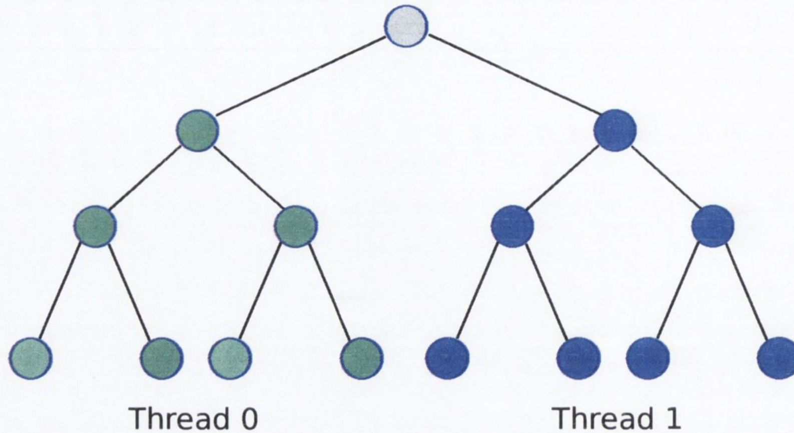


Figure 3.4: *Organisation of threads in the Subtree Builder.*

Multithreaded SAH Calculation

Once the partitioning units pass all of their primitives into the binning units, they must wait in state *S2* for each of them to be binned and for an SAH calculator to return the next split so that they can begin partitioning again. In an implementation such as the one presented in this work, the total combined latency of the binning and SAH calculators can be of the order of 100 cycles or so. Stalling frequently in state *S2* would represent a large performance penalty because it would be incurred on every internal node in the tree. To combat this, a multithreaded approach is taken.

Context for multiple threads is maintained in the system, as shown near the top of Figure 3.2. Each thread represents a different path in the tree. Initially, there is only one thread in the system, representing the root node of the hierarchy. As new child nodes are created, they spawn new threads, until a maximum number of threads is reached. Each thread context stores the ranges in each of the primitive buffers of the primitives in the thread, a split position and axis, a stack and stack pointer and a node AABB. Each thread context (minus the stack) is relatively small ($<$ 50 bytes), so they are stored in registers inside the partitioning units. The new threads represent different subtrees, as illustrated in Figure 3.4. Multiple SAH calculator pairs are visible in Figure 3.2, as each new thread is statically assigned to one of these pairs.

When a partitioning unit finishes partitioning a node, instead of stalling for the SAH calculation in state *S2*, it can switch context to the next thread in the system. It then proceeds through the flowchart in precisely the same manner. Once it has completed its work for the last thread, it can return to the first thread for which the split will often be ready. Each partitioning unit is free to switch threads asynchronously of the other partitioning units. However, since all partitioning units contribute to processing each node in the hierarchy, a partitioning unit cannot proceed with a thread until all other partitioning units have computed their respective contributions towards determining the split for the next node in that thread.

3.3.2 Partitioning Unit Datapath

Along with coordinating the flow of data through the subtree builder, the partitioning unit also implements datapath components for determining on which side of a given SAH split a primitive resides.

Figure 3.5 shows a detailed diagram of the partitioning unit datapath, which is under the direction

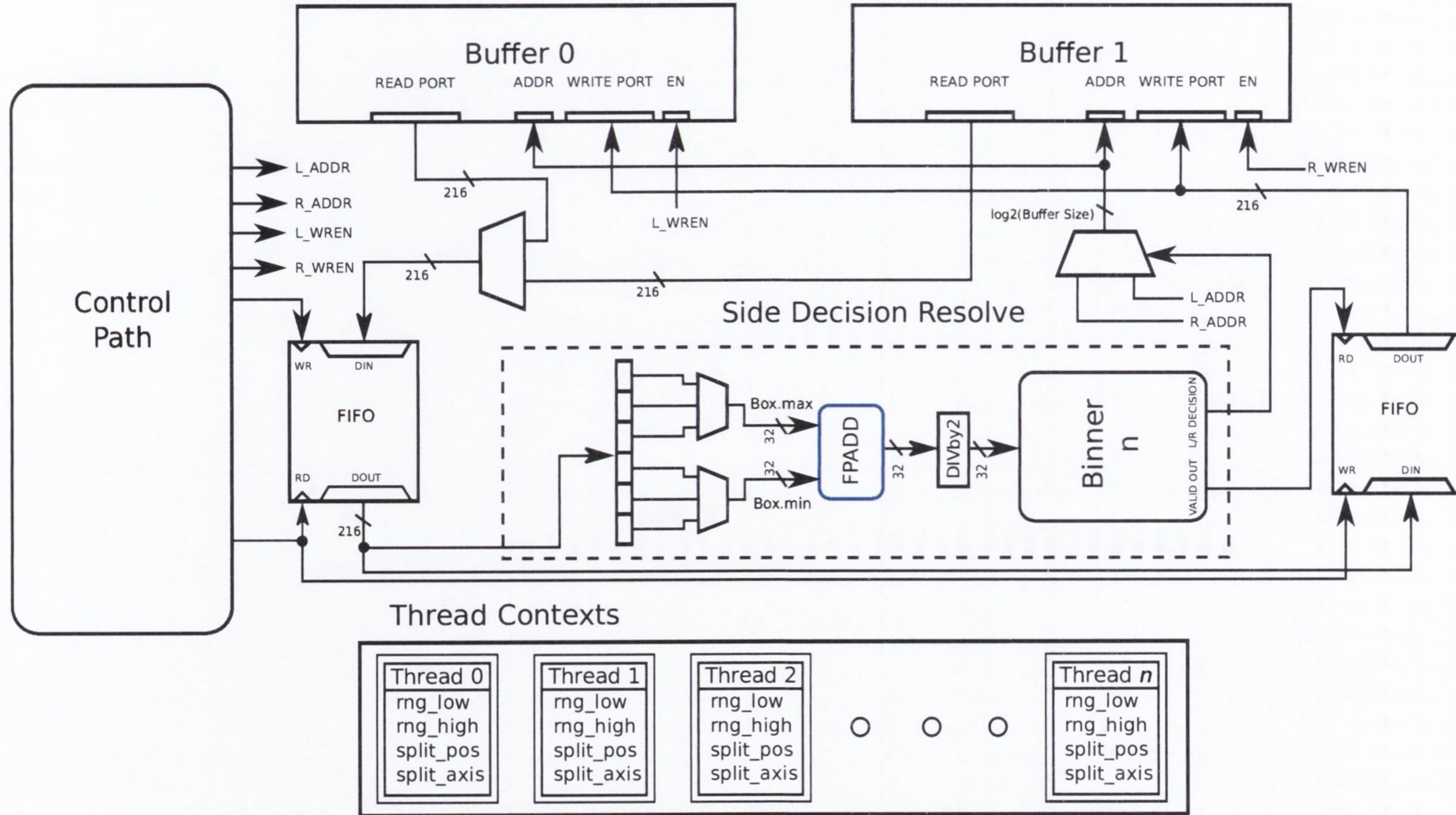


Figure 3.5: *Detailed schematic of the partitioning unit datapath. Simplified portions of the control path are shown to display interfaces and to give context.*

of the control path, shown on the left hand side of the figure. The core of the partitioning unit datapath is the logic which determines on which side of the split a given primitive AABB lies. This logic is marked *Side Decision Resolve* in Figure 3.5, and is fully pipelined. When a partitioning operation begins, primitives are read from the buffers. Since source primitives can arrive from either buffer in the primitive buffer pair, a multiplexer is used to decide which port to read from and enable signals are used to decide which buffer to write to. As they are read from the buffer, they are placed in a FIFO. Primitives are read from this FIFO and routed through the side decision logic which first calculates the centre of the AABB in the given split axis and determines on which side of the split the centre resides.

As each thread at any given time is operating on a different node of the hierarchy, fine-grained temporal multithreading is implemented in the partitioning unit. Data from individual thread states (shown at the bottom of Figure 3.5) is multiplexed in at each stage of the pipeline, depending on which thread is active in that stage. This allows many threads to be present in the pipeline at once, improving overall throughput of the partitioning operations.

To write out the newly partitioned list of primitives, two address registers are used for each thread, *rng_low* and *rng_high* which are placed at the beginning and end of the primitive list at the start of processing. As primitives pass through the pipeline, any primitive on the left of the split is placed in location *rng_low*, and the *rng_low* address register is incremented. If the primitive resides on the right side of the split, the *rng_high* address register is used and decremented after the write. Using the *rng_low* and *rng_high* registers in this way means that the set of primitives from one buffer ends up in the same address range in the opposite buffer in new partitioned order.

When a partitioning unit finishes one thread, it immediately (within two cycles) switches to the next thread. When primitives from this new thread begin passing through the datapath, new values such as the chosen split axis and node AABB will be multiplexed into individual pipeline stages as appropriate. This process will continue until all threads have been processed, at which point the partitioning unit will halt, and wait for a signal from the upper control logic that a new split is available for it to begin processing again.

As primitive AABBs are written back into the buffers in their new partitioned order, they are also being passed concurrently to the binning units (Figure 3.2) so that the splits for the two child nodes can be calculated. As this occurs in parallel, by the time the partitioning units have finished the current iteration for all threads, the new split for Thread 0 may already be available. In this case, the partitioning units can continue processing all threads from the beginning without stalling.

3.3.3 Binning Unit

The binning units are responsible for determining the bin positions for each primitive AABB that must be binned into the current node. While the partitioning units are rearranging the primitive AABBs to perform a split of a parent node, the binning units collect the primitive AABBs in the same order and bin them into either the left or right child node AABB, depending on which child the primitive belongs to. This is possible because after a split is chosen for a node, the AABBs of the resulting child nodes are known. To determine the bin position, Bin_n, of a primitive P in a given axis n, the centre C of the primitive is first calculated. If S is the number of bins to be used to sample the SAH, the formula for determining the bin position of the primitive in node bounding box BB is given by:

$$Bin_n = \frac{C_n - BB.min_n}{BB.max_n - BB.min_n} \times S \tag{3.1}$$

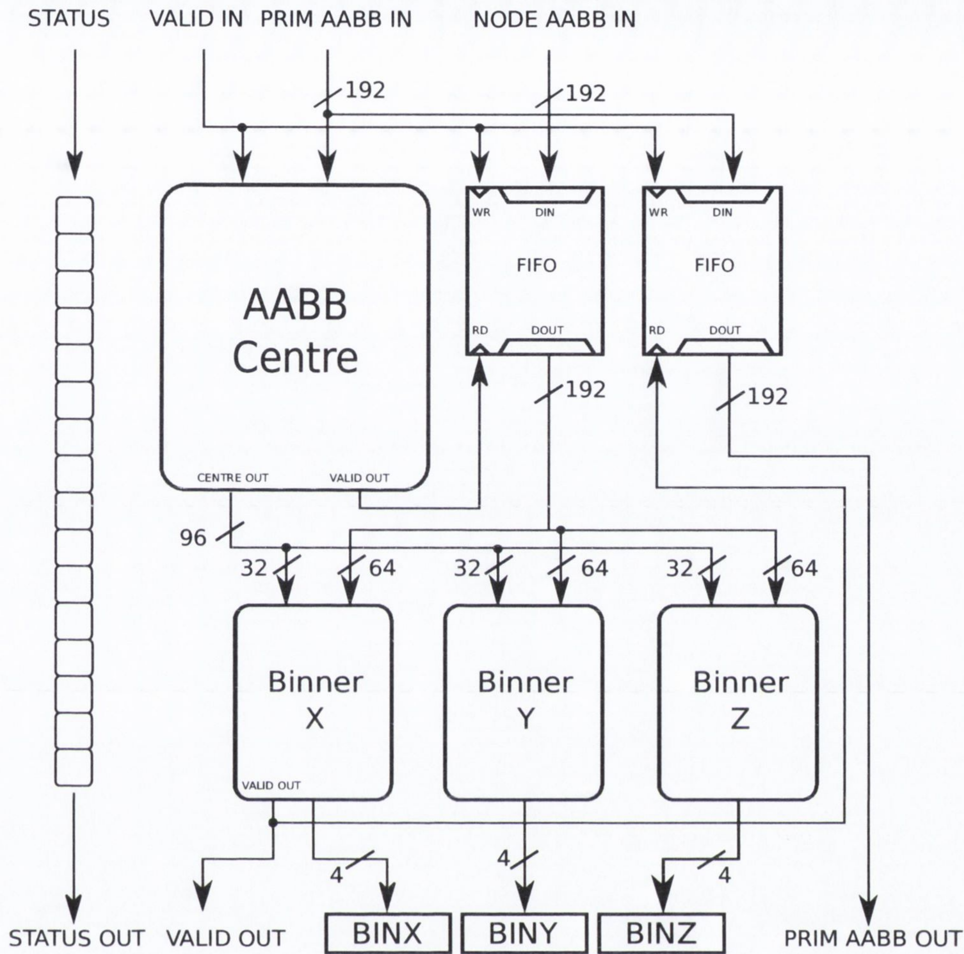


Figure 3.6: *Top Level Schematic of a Binning Unit group, including auxiliary logic.*

The floating-point value obtained from the formula is converted to an integer to yield the bin ID. To avoid invalid bin values due to numerical precision problems, the result is capped to the valid bin range. The output of a binning unit is the bin position of the primitive AABB in one axis, as well as the primitive AABB itself. Figure 3.2 shows how binning units occur in groups of three, so that the bin position of the primitive can be obtained in all three axes. Bin decisions and primitives are then fed to the SAH calculators which use this information to update both a primitive counter and AABB for each bin in each axis. All binning units in a group operate in parallel and share temporary storage for the original primitive AABB while its bins are being calculated.

Figure 3.6 shows how a group of three binning units is instantiated in the design, coupled with auxiliary logic. Each group takes a primitive AABB and node AABB as input, and provides the bin position in all three axes. Accompanying each group of binning units are various auxiliary units. Visible in the upper half of this diagram is an *AABB centre unit*, which calculates the centre of each incoming primitive AABB, and which is shared between the three binning units. To the right of the AABB centre unit are two FIFOs. The leftmost FIFO serves as temporary storage for the current hierarchy node's AABB (*BB* in Equation 3.1), which will be used once the AABB centre unit finds the centre C so that

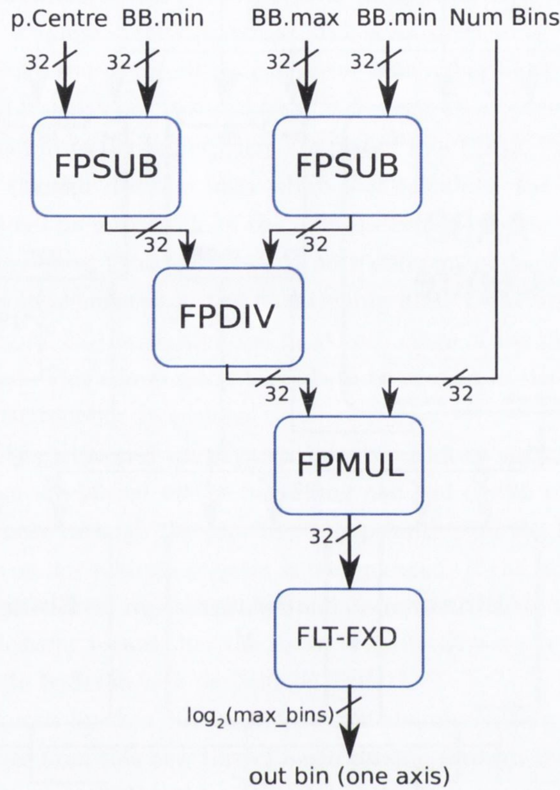


Figure 3.7: *Internal structure of* Binner X,Y,Z *shown in Figure 3.6. Each binning unit performs the binning of a primitive AABB p into the node AABB BB for one axis. A* FLT-FXD *stage is needed to convert the floating-point bin value to an integer which can be used to address the bin to be updated.*

Equation 3.1 can be evaluated. The rightmost FIFO serves to store the primitive AABB such that it can be output to the SAH calculators once the bin positions of the primitive have been determined. These FIFOs are also shared between the three binning units.

Once the centre of the primitive AABB has been calculated, it is passed down to the three binning units in the lower half of Figure 3.6. There are three identical modules, one for each of the three axes. A detailed internal diagram of each of these modules is shown in Figure 3.7. The function of these units is to evaluate Equation 3.1. The output of one of these units is the bin position in the unit's assigned axis (X,Y or Z). When a valid result exits these units, a read signal is sent to the rightmost FIFO, which outputs the original primitive AABB along with the bin positions. This information is then passed on to the SAH calculators.

On the left hand side of Figure 3.6, a status shift register can be seen. The purpose of this shift register is to record various items of information concerning the operands present in each pipeline stage of the binning units. Like partitioning units, the binning units and their auxiliary units are also fully pipelined, and allow for operands from different threads to be processed in different pipeline stages in any given cycle. The status shift register facilitates this by keeping track of which thread is present in each stage, as well as storing various other low-level control signals.

3.3.4 SAH Calculator

Figure 3.8 shows a top-level diagram of the SAH calculator. At any one time, each SAH calculator is working to determine the lowest-cost SAH split position and axis for a single node. The SAH computation flows from the top of the figure to the bottom, where the final SAH split information can be seen on the output. The very first stage of the SAH calculation is performed by the *SAH Bin Collectors*, shown at the top of the figure. The purpose of an SAH Bin Collector is to accumulate, for each SAH bin in each axis, an AABB and the number of primitives residing in that bin. Since each group of three binning units are operating on a subset of the primitives in any given node, there exists one SAH Bin Collector per binning unit group, and binning unit groups are hardwired to an SAH Bin Collector. The purpose of this is to allow binning units to pass data into the SAH calculators without contention, as binning units will be operating on their assigned subsets of node primitives in parallel. Each SAH Bin Collector thus accumulates only a partial set of AABBs and primitive counts that correspond to its subset of primitives.

Once a partial vector for each axis has been generated in each SAH Bin Collector, the vectors are then merged into the final lists. To achieve this, the *SAH Bin Merge* units are utilised. When all primitives have been accumulated, control logic inside the SAH calculator instructs each SAH Bin Collector to dump its full contents into the SAH Bin Merge units. There is one SAH Bin Merge unit for each axis, and all operate in parallel. Each SAH Bin Merge unit accepts one partial vector from each SAH Bin Collector, according to which axis it is responsible for. For example, the unit labelled "SAH Bin Merge X" in Figure 3.8 will accept a partial X-axis vector from each SAH Bin Collector. Each SAH Bin Merge unit combines the partial vectors from the SAH Bin collectors to produce the final stream of AABBs and primitive counts for its assigned axis.

Once these streams are available, the *SAH cost evaluators* are activated. There exists one SAH cost evaluator per axis, and each axis is evaluated in parallel. Each SAH cost evaluator takes a stream of AABBs and primitive counts from the SAH Bin Merge unit it is connected to, and evaluates the SAH formula for each bin in the stream, ultimately outputting the lowest-cost SAH split for its axis, as well as the AABBs and primitive counts of the resulting child nodes.

Finally, the last stage of the SAH calculator is activated. The lowest-cost SAH splits found for each axis are then compared, and the final axis and split are chosen ("MUX Lowest Cost" at the bottom of Figure 3.8). All relevant information, such as the child node AABBs and primitive counts are multiplexed though this stage. Once the split has been determined, the SAH calculator raises a "READY" signal, which indicates to the partitioning units that this node is ready for splitting.

SAH Bin Collector

The SAH Bin Collectors are responsible for accumulating the vector of AABBs and bin counts. In order to achieve this, the SAH Bin Collectors are themselves constructed of a set of three *SAH uni-collectors*. Each SAH Bin Collector contains three Uni-collectors; one for each axis. One SAH uni-collector is pictured in Figure 3.9. SAH uni-collectors utilise a small buffer which stores a 216-bit vector for each SAH bin. The lower 192 bits of this vector store an AABB and the upper 24 bits store a primitive count (Figure 3.10). Primitive AABBs, along with their chosen bin positions are passed into the SAH uni-collectors from the binning units. The chosen bin is then used as an address to the buffer and to fetch the current state of that bin. When the bin has been fetched, the new primitive AABB and the AABB from the buffer are together passed into an AABB accumulation unit which calculates the union of the two, resulting in the new AABB for the bin. Similarly, the existing primitive count contained in the word read from the buffer

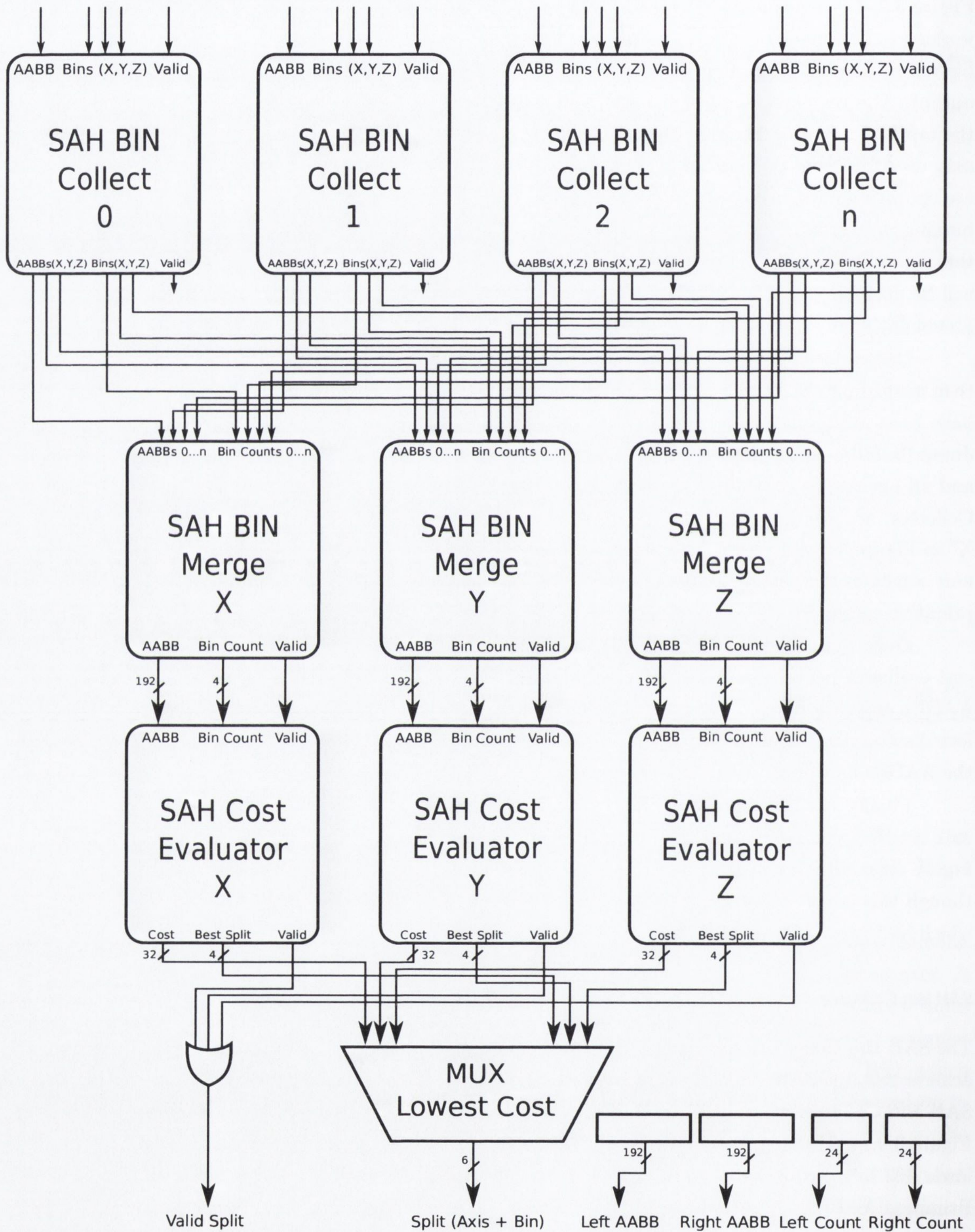


Figure 3.8: *Top-level view of the SAH Calculator.*

is incremented. The updated 216-bit vector is then written back into the buffer.

The SAH Bin Collector is designed to accept a new primitive on every cycle. In real circuits, buffers typically show a latency of some number of cycles. In the current implementation, these buffers have a latency of one cycle, which means data arrives on the read port the cycle after it is addressed. To deal with the constant stream of primitives into the unit, a two-stage pipeline is employed. The first stage reads the current state of the bin from the buffer and the second performs the update and write back. This results in a possible data hazard, as if a fetch and an update are performed to the same location on the same clock cycle, outdated operands are obtained in the fetch. To overcome this, additional logic is employed to cache the most recently updated location in a register, and if the pipeline detects an access to the same location twice in a row, the cached value is used instead of the previously read value. This is similar to *operand forwarding* in CPU pipelines, and eliminates stalling on account of the buffer latency, allowing for maximum throughput.

When the list of bins in each buffer has been updated with all primitives, the bins are output consecutively from the Uni-collector using the *dump state machine* shown near the bottom left of Figure 3.9. This forwards the bins to the next set of units in the SAH calculator.

SAH Bin Merge Unit

The SAH bin merge units shown in Figure 3.8 are the simplest unit in the SAH calculator. There is one SAH bin merge unit for each axis. The input to each SAH Bin Merge is one partial stream of primitive AABBs and counts from each SAH Bin Collector. The SAH bin merge units combine these partial vectors via logarithmic reduction, by employing a tree of AABB union operators to produce the final AABB for each bin, and a tree of integer adders for the primitive counts. Finally, they output the final bin stream for their assigned axis.

SAH Cost Evaluator

Figure 3.11 shows a detailed diagram of the SAH cost evaluator. The input to the SAH cost evaluator is a stream of AABBs and primitive counts, each representing a bin in the binned SAH calculation. For each element in the stream, the SAH cost evaluator computes the SAH formula. This formula is restated for convenience:

$$C(V \rightarrow \{L, R\}) = K_T + K_I \left(\frac{SA(V_L)}{SA(V)} N_L + \frac{SA(V_R)}{SA(V)} N_R \right) \tag{3.2}$$

In this form, V is the original volume, V_L and V_R are the subvolumes of the left and right child nodes, N_L and N_R are the number of primitives in the left and right child nodes, and SA is the surface area. K_I and K_T are implementation-specific constants representing the cost of ray/primitive intersection and traversal respectively. For finding the lowest cost, the constants in the equation can be removed, as well as the common divisor $SA(V)$ of the remaining terms, as doing so does not affect the relative magnitudes of the costs. Therefore, the SAH cost evaluator uses the well-known simplified expression:

$$C(V \rightarrow \{L, R\}) = SA(V_L)N_L + SA(V_R)N_R \tag{3.3}$$

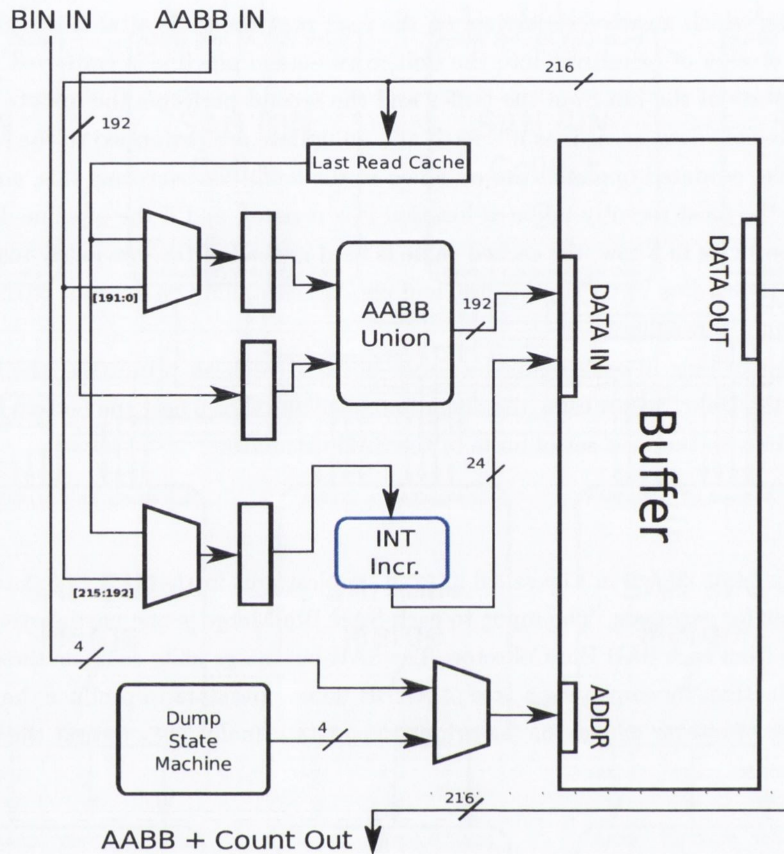


Figure 3.9: *Architecture of the SAH Uni-collector. This unit is responsible for accumulating SAH bin AABBs and primitive counts in a single axis for a given stream of primitives. Once the stream is accumulated, the unit can dump the accumulated vector to its output using the dump state machine.*

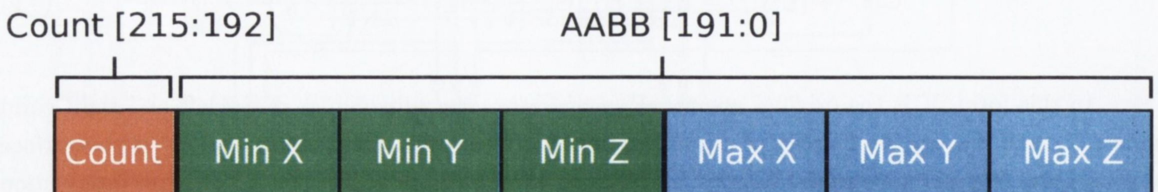


Figure 3.10: *Data layout of the SAH bin record in the SAH Uni-Collector internal buffer. The record is 216 bits in size. The lower 192 bits are used to store a 32-bit floating-point AABB and the upper 24 bits are used to store the number of primitives in the bin. One such record is needed for every bin used to calculate the SAH split.*

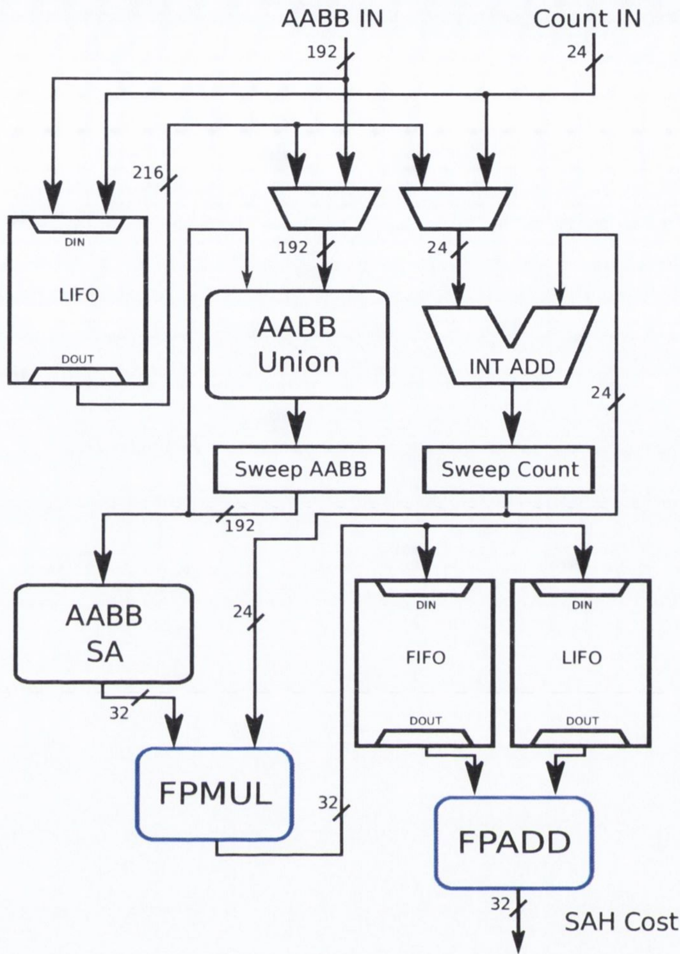


Figure 3.11: *The SAH cost evaluator takes an input stream of bin AABBs and primitive counts and performs a bidirectional sweep through the stream to obtain both terms needed for Equation 3.3. The two terms are ultimately added together to get the final SAH cost of each split candidate in the stream.*

To obtain all terms of the equation for each split candidate, it is necessary to know the union of all bin AABBs and the sum of all primitive counts left and right of each split candidate. To achieve this, the unit first calculates all the $SA(V_L)N_L$ terms for all split candidates. The SAH cost evaluator receives the list of AABBs and primitive counts in left to right order. The list is immediately passed into the AABB union and integer adder units shown near the top of Figure 3.11. Visible underneath the AABB union and integer adder units are registers labelled "Sweep AABB" and "Sweep Count". These registers keep track of the union of all AABBs and the sum of all primitive counts in the stream so far. A feedback loop exists between this register and the AABB union and integer units to implement this accumulation. This yields the V_L and N_L terms in order from left to right.

As the new stream of terms is leaving the AABB union unit and adder, the surface area of each AABB is calculated in turn by the unit labelled *AABB SA*, near the left hand side of Figure 3.11. The output of the integer adder is delayed by the same number of cycles as it takes to calculate the surface area of the AABB. Once ready, both are sent to a floating-point multiplier. The output of the multiplier yields the $SA(V_L)N_L$ terms. These terms are then stored temporarily in a FIFO for later use, as shown in the figure.

As the $SA(V_L)N_L$ terms are being calculated, the raw, unmodified input stream is also stored in the LIFO show near the top left of the diagram. The purpose of the LIFO is to help calculate the $SA(V_R)N_R$ terms. Reading from this LIFO provides the original input stream in reverse order (from right to left).

Once all the $SA(V_L)N_L$ terms have been calculated and stored in the FIFO, the original input stream is read from the LIFO and is passed through the same data path to obtain the $SA(V_R)N_R$ terms. As these terms begin to emerge, they are placed in a second LIFO, to the right of the FIFO in the diagram. By reading the $SA(V_L)N_L$ and $SA(V_R)N_R$ terms from the FIFO and LIFO respectively, it is possible to calculate the full SAH cost of each split candidate in order by passing the terms into a final adder circuit, shown near the bottom of the figure. The unit keeps track of the lowest-cost split found so far. Once the cost for each split has been calculated, the unit outputs the lowest-cost split candidate in its assigned axis.

3.3.5 Tree Output and Format

The last major component of the subtree builder is the logic which handles the BVH tree output. Any design for such logic must first establish the storage format and data layout of the BVH data structure in memory. The AABB-index pair primitive format elected for use with the subtree builder naturally lends itself to a BVH storing indices at its leaves, as opposed to the primitives themselves. It was therefore decided to utilise this format for the contents of the BVH leaves. The first responsibility of the tree output logic is to produce a compact list of primitive indices which leaf nodes of the BVH can index.

Indices Output

Before construction of the subtree begins, the subtree builder's primitive buffers are first loaded with primitives. Since many pairs of primitive buffers can exist, primitives may be distributed over multiple pairs, and under the control of several different partitioning units. In fact, as each node in the hierarchy is constructed in a data parallel manner across multiple partitioning units, indices within a given node are likely to be distributed over several buffers. Although this approach provides a way to parallelise the computation, it complicates tree output, as it is necessary to recombine several (and often a variable number) of indices from multiple sources which are operating somewhat asynchronously.

To solve this problem, a simple scheme was devised. Coupled with each partitioning unit in the design is a set of *index output FIFOs*, one for each thread present in the system. In terms of the design constants defined near the beginning of this chapter, the total number of such FIFOs in the subtree builder is therefore $w \times Tsize$. When a partitioning unit encounters a leaf, if it controls any primitives contained in this leaf, it will write them into its index output FIFO that corresponds to the thread that the leaf belongs to.

Each of these indices is stored in the FIFO along with two additional bits. The first bit is a *valid bit*, and indicates if the index is a true index, or whether it is a "null" index. The second bit is the *last bit*. The last bit constitutes a mechanism by which a partitioning unit can signal that this is the last index that it possesses for this particular leaf. As the build progresses and multiple leaves are encountered, it is possible for indices belonging to multiple leaves to be present in an index output FIFO at any given time. The last bit allows the tree output logic to delineate between indices belonging to these different leaves.

Figure 3.12 illustrates how index output FIFOs are used to reorder the indices for tree output. The figure illustrates how index output occurs for a single thread, and a copy of this setup will exist for each thread in the subtree builder. Two FIFOs are shown to serve as a clear example of how the scheme operates, but the number of such FIFOs in a subtree builder will be greater in most instantiations. As

each partitioning unit possesses its own FIFO for each thread, the two FIFOs shown in the figure belong to different partitioning units, but contain indices from the same thread which must be recombined. Below the FIFOs is the logic which performs this recombination. It achieves this by taking advantage of the fact that since partitioning units cooperate to construct a leaf, they will visit each leaf node in the same order, and thus groups of indices will be in the same "leaf order" across the two FIFOs. Furthermore, valid and last bits delineate indices from different leaves.

The selection logic collects indices from these FIFOs one leaf at a time, by keeping track of the last bits of the indices. At the beginning of a new leaf, the selection logic will read primitives round-robin from the FIFOs. If upon reading an index it encounters a last bit, it will stop reading from that FIFO until it has encountered a last bit from all the other FIFOs. This ensures that the reading of indices belonging to future leaves is postponed until all indices from the current leaf have been read.

This scheme depends on the fact that a last bit must be received from each FIFO. However, there will be a reasonable chance that one or more partitioning units will not be in control of any primitives in a given leaf, especially when dealing with small leaf sizes. To address this case, a partitioning unit can write a "dummy" index into its FIFO, with the last bit set and the valid bit unset. This signals to the selection logic that the partitioning unit associated with this FIFO does not posses any primitives contained in this leaf, but that it is ready to proceed with the next leaf. The selection logic will thus discard the fake index and record this fact in its internal state. Indices are read through a multiplexer one at a time from the group of FIFOs, and deposited into a 24-bit $\times$ n shift register. This shift register collects indices into a group of n so as to match the output memory data width. One copy of this selection logic exists for each thread in the system. To write out indices to memory, the subtree builder multiplexes the output of the shift registers into a final output FIFO.

At the beginning of construction, the subtree builder must be provided with an address to begin writing indices. Since the required memory size for indices is predictable, dynamic memory allocation is not needed. In a full system, it is the responsibility of the logic calling upon a subtree builder to ensure that sufficient space is available beginning at the provided address to write all of the indices.

BVH Node Output

The second component of the tree output is the hierarchy nodes themselves. A simple format was chosen for the nodes, and consists of two 192-bit words per node. The first word stores a standard 32-bit floating point AABB. The second word of the node contains two 32-bit pointers which represent pointers to its child nodes. One additional bit in the second word is used to signify if the node is a leaf or not. In the event that the node is a leaf, the first of the 32-bit pointers is used to store an index into the list of primitive indices, and the second 32-bits are used to store the number of primitives in the leaf. The rest of the second word (127 bits) is padded to simplify the hardware design. This format thus results in a BVH node data structure 384 bits in size, which is still reasonably compact.

The node output is in some ways simpler than the indices output, and in other ways is much more complex. Although multiple partitioning units cooperate to construct a node, the required information for assembling the BVH node data structure, such as the AABB, has already been combined from multiple sources within the subtree builder itself and is available in existing registers in the system (e.g. in the thread state registers). In addition, the sequencing of when to write out the node can largely be derived from monitoring the output signals of the various units; for example, monitoring "DONE" and "READY" or "GO" signals in the partitioning units and SAH calculators is sufficient to infer when the tree output

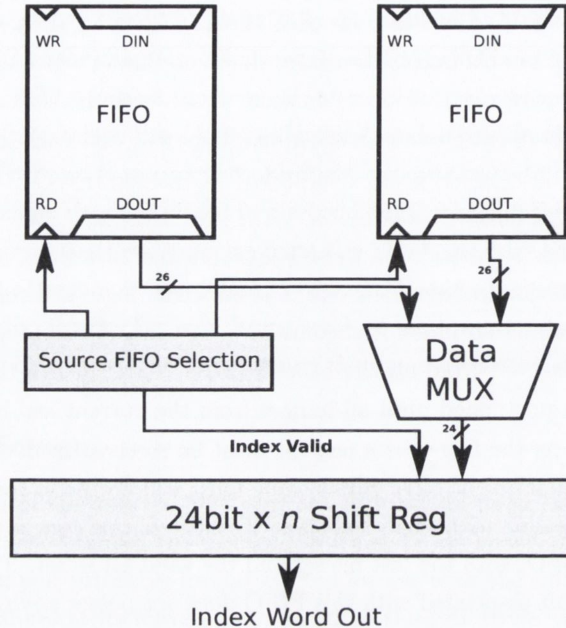


Figure 3.12: *Schematic of the index selection logic showing multiplexing of two index sources associated with different partitioning units. The selection logic decides which source FIFO to read from next based on a simple scheme utilising two additional bits per entry in the FIFOs. The multiplexer reads an index from the source FIFO and places it into a $n\times 24$-bit shift register. This shift register is used to group indices together into data words that match the memory interface of the subtree builder.*

logic can safely latch the contents of these existing registers for the purposes of writing out a node.

Although indices are simply written as a consecutive list from the subtree builder, the BVH node data layout is slightly more complex. As the subtree builder uses strict depth-first construction ordering, a depth-first tree output is chosen also. During construction, when a node is written out, construction proceeds with the left child, and the right child is pushed to the stack. One difficulty with doing this is that if neither child node resulting from a split represents a leaf node, construction must proceed with one of them. It is not possible to know in advance how many nodes will be created before the other child will be popped from the stack and written out (although the value is bounded by the remaining number of indices in the subtree). To deal with this, the AABB of any internal node is first written out, and the address is incremented by two, leaving a gap which will later be filled in by the child pointers. Simultaneously, the address of this gap is pushed to a stack. Gap addresses are continually pushed to a stack until a leaf node is encountered. In the case of a leaf, the stack is not needed and the pointers may be written out immediately along with the node AABB. After writing the leaf, the stack is popped and since the next address represents the location needed to complete the child pointers at the popped address, the child pointers can now be written out, filling the gap left earlier.

The second uncertainty with writing nodes is how many nodes will be written in total. For the BVH, this value is bounded by $2N-1$ where N is the number of primitives referenced by the BVH. However, in order to avoid wasting memory, a dynamic memory allocation scheme is built into the subtree builder. While writing child nodes to memory, the subtree builder will request *blocks* of a fixed size from an external source via its output ports. In a real system utilising the subtree builder, it is the responsibility of logic outside the subtree builder to respond to these requests, and to ensure that the blocks provided

to the subtree represent empty memory. The node output logic will write nodes consecutively into its currently assigned block, requesting a new block once it has exhausted its allocated space.

3.4 System Integration and the Upper Builder

The subtree builder represents a hardware device which is capable of constructing BVHs for a set of primitives up to a maximum size. Given that each primitive in the builder consumes 27 bytes of storage, even a few thousand primitives begins to consume a considerable quantity of memory, which would lead to prohibitively large primitive buffers for large scenes. The subtree builder is therefore by itself not capable of constructing BVHs for the kinds of scenes which modern interactive ray-tracers can render at interactive rates (such scenes are typically of the order of several hundred thousand to millions of primitives). This is not a problem, as the subtree builder is not designed to achieve this, but is a component intended to be included in a modular system alongside other components.

Ultimately, it is envisioned that the kind of hardware described in this thesis will be useful to future graphics processors. In its current form, the BVH construction hardware is most useful to ray-tracing processors specifically, but the broad importance of spatial index structures laid out in Chapter 1 motivates its use in more conventional GPUs.

The decomposition of BVH construction into several components, one of which being the subtree builder, provides a number of options for how to utilise the subtree builder in a real processor. Many architects propose that future processors will take advantage of heterogeneous architectures incorporating conventional multicore processors, stream multiprocessors such as GPGPUs, and also custom logic to achieve efficiency by distributing different computations to the most suitable subsystems [CMHM10]. Such a computing model is already visible in a number of processors, including mobile processors such as NVIDIA Tegra [TMCS08], and even the recently proposed SGRT mobile ray-tracing processor [LSL*13]. Both of these systems combine conventional multicore, manycore and fixed-function components on the same die. Figure 3.13 shows the basic model of how hardware BVH construction may be integrated into such a processor.

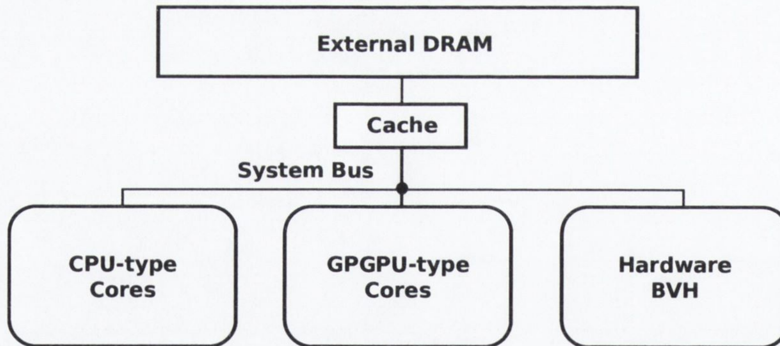


Figure 3.13: *System integration model. The hardware BVH is connected to programmable cores to comprise a heterogeneous computing system.*

The subtree builder is capable of constructing only binned SAH BVHs, but incorporating a subtree builder in a processor in this manner leads to quite a flexible BVH construction system. The programmable components could be leveraged to construct the upper levels of the hierarchy, while relying on the subtree builder to construct the lower levels, allowing full BVH construction for large scenes. It is likely that this approach would still realise the benefits of the custom BVH hardware to a large extent, as the majority of the workload even in a large scene will occur in the lower levels of the hierarchy. Other possibilities for utilising one or more subtree builders in this kind of system include:

- Many renderers utilise a combination of rebuilding the BVH and refitting/updating mechanisms. By coupling the subtree builder with programmable components in this way, such an implementation would be trivial. This would particularly hold true for schemes such as selective restructuring [YCM07], which identifies subtrees in the hierarchy for rebuilding, while refitting others. Typically, subtrees near the bottom of the hierarchy will undergo the most perturbation during scene motion, and thus the subtree builder would likely incur the greatest fraction of rebuilding duties under such a scheme.
- Lazy hierarchy construction would likely be straightforward to implement. Programmable components could identify subtrees which are actually visited by rays, and request construction of these subtrees from the subtree builder.
- Programmable components could also act as a "scene pre-processor" by implementing, for example, triangle pre-splitting using techniques such as that proposed by Ernst and Greiner [EG07], or techniques which build an SAH BVH over clusters of primitives, such as occurs in HLBVH-style builders [PL10, GPM11], or in pre-clustering methods such as the technique proposed by Garanzha [Gar09]. Many such techniques require little or no modification to the actual hierarchy construction, as, like scene primitives, clusters are typically also represented as AABBs. Moreover, these techniques often reduce the number of primitive elements involved in the build, potentially allowing the entire hierarchy construction portion of the algorithm to occur on-chip in the subtree builder.

A further interesting observation of integrating the subtree builder in this manner is that existing on-chip memory resources, such as the cache, could be utilised in place of the primitive buffers inside the subtree builders if desired. Interestingly, the recently proposed STRaTA architecture [KSS*13] builds upon the TRaX traversal architecture by specialising a portion of the L2 cache for storing ray streams. Taking a more tightly integrated approach to the system in this manner could improve efficiency by eliminating the hardware cost of dedicated buffers for the subtree builder.

3.4.1 Upper Builder

Although coupling the subtree builder by itself into a heterogeneous processor would allow for full hierarchy construction and support a variety of construction techniques, it may be desired to construct the hierarchy entirely within a fixed-function BVH construction subsystem. To achieve this, subtree builders must be accompanied by logic to construct the upper levels of the hierarchy. This logic must be capable of interacting with external memory in order to deal with the larger working set involved in constructing upper nodes. As described at the beginning of Section 3.2, such logic was implemented as part of this project, and is termed the *Upper Builder*.

The upper builder is responsible for performing exactly the same operations as the subtree builder, but for upper hierarchy nodes. This naturally leads to a component which is very similar to the subtree builder. The upper builder also contains partitioning units, binning units and an SAH calculator pair which are only modified slightly from their counterparts in the subtree builder. The principal differences between a subtree builder and the upper builder are as follows:

1. The upper builder reads and writes primitives from/to external memory in lieu of accessing on-chip primitive buffers. It includes additional logic for efficiently utilising modern DRAM systems, by reading primitives in bursts and buffering writes into bursts before they are requested.

2. The upper builder contains no multithreading support (only one thread is active in the upper builder). Multithreading is unnecessary for the upper builder because it only constructs the uppermost nodes of the hierarchy, which usually contain thousands of primitives at least, and which are read in long streaming consecutive reads. Therefore the stall incurred by waiting on the binning units and SAH calculators represents a negligible fraction of the overall execution time. Implementing multithreading in the upper builder would spend hardware resources while garnering almost no performance benefit.

3. Alongside constructing the upper levels of the hierarchy, the upper builder is responsible for providing subtrees to any subtree builders it is connected to. When the upper builder encounters a node which encompasses a set of primitives that will fit inside the primitive buffers of a subtree builder, it fills the subtree builder with the primitives and issues a 'GO' signal to the subtree builder.

4. As described in Section 3.3.5, subtree builders must be provided with an address to begin writing indices to. Additionally, they periodically request blocks of memory in which to write hierarchy nodes. The upper builder provides these services. The upper builder maintains a simple memory pool and arbitrates between potentially simultaneous block requests from multiple subtree builders.

3.5 Implementation

To evaluate the architecture described in this chapter, it was implemented as a VHSIC Hardware Description Language (VHDL) model using only the synthesizable subset of the language. The model is written as synthesizable register transfer level (RTL) code (i.e. on the level of individual arithmetic units, shifters, registers, multiplexers, wires etc.). Each module in the system is implemented as a separate VHDL entity.

3.5.1 Primitive Modules

To implement such a large and complex design at this level, library modules are required for the various floating-point units, SRAMs and register files as shown in schematics through this chapter. For these purposes, a library of configurable modules was created by the author and used throughout the design. Table 3.1 summarises the characteristics of the floating-point (FP) units utilised throughout the design. All floating-point cores are IEEE-754 compliant [IEE08]. Similarly, Table 3.3 summarises the memory components used in the design. Unless otherwise stated, these characteristics apply system-wide. All units are fully pipelined. *Latency* refers to the pipeline length of the unit. A *throughput* of n indicates that a result may be obtained every n cycles.

IP type	Latency	Throughput	Description
FPADD	4	1	32-bit IEEE-754 compliant floating-point adder.
FPSUB	4	1	32-bit IEEE-754 compliant floating-point subtracter.
FPMUL	4	1	32-bit IEEE-754 compliant floating-point multiplier.
FPDIV	12	1	32-bit IEEE-754 compliant floating-point divider.
FPCMP	1	1	32-bit IEEE-754 compliant floating-point comparator.

Table 3.1: *The essential characteristics of the floating-point (FP) modules used in the implementation.*

IP type	Latency	Throughput	Description
SRAM	1 or 2	1	Fast internal SRAM, various sizes.
FIFO	3	1	First-In, First-Out Buffer, various sizes.
LIFO	1 or 2	1	Last-In, First-Out Buffer, various sizes.

Table 3.2: *The essential characteristics of the memory resources used in the implementation.*

Unit	Latency	Throughput
Primitive Buffer	2	1
Partitioning Unit	36	1
Binning Unit	32	1
SAH Calculator	73	73

Table 3.3: *The essential characteristics of the higher-level units in the design.*

3.5.2 Higher-Level Modules

Given these characteristics for primitive hardware components in the design, the following characteristics of the higher-level modules are obtained. The SAH Calculator is the only unit which has not been fully

pipelined, and therefore has a much lower throughput. Pipelining of the unit would be possible, and would likely lead to a more efficient architecture, but this is left to future work.

3.5.3 DRAM Model

To model DRAM interfaces, a generic Verilog DDR model from DRC computer was used. This DDR model provides an interface with address and data lines, a read/write signal, burst length etc. Each memory channel at peak is capable of delivering one 192-bit word per cycle and supports burst and single transactions. Simulations were performed assuming that each memory channel could always provide data quickly when requested. This results in a somewhat "ideal" model of a real DRAM chip interface, as real systems are subject to a number of factors which reduce their real-world performance relative to their theoretical maximum. However, as will be seen in Chapter 4, the bandwidth supplied from the DRAM ports to the BVH hardware is only a fraction of the bandwidth available to most modern processors (less than a quarter in some cases). Furthermore, the BVH hardware for the most part generates highly contiguous access patterns (sweeps through memory) of the same type (either reads or writes), which is an ideal case for modern DRAM chips for reaching close to their theoretical maximum performance [RDK*00]. For these reasons, it is highly likely that a real system could supply a similar effective bandwidth to that provided by the simplified model. It was thus reasoned that the simplified model was sufficient to give reliable results.

3.5.4 Software and Testing

A variety of software was used to carry out the experiments detailed in Chapter 4.

- **Questasim RTL Simulator**

  To carry out RTL simulations of the architecture model, the Questasim RTL simulator from Mentor Graphics was used [Men12]. Versions 6.6 and 10.0 were used throughout the project.

- **InCyte Chip Estimator** To perform area and power estimates of the BVH hardware, the free version of the InCyte Chip Estimator tool (version 6.1.0) from Cadence was used [Cad14]. This tool takes a collection of higher-level statistics of a design as input (including clock frequency, technology node and number and type of hardware IP cores used in the design) and makes accurate area and power estimates based on these.

- **Leonardo Spectrum** To perform small auxiliary logic synthesis tasks to assist in the InCyte chip estimations, the Leonardo Spectrum ASIC synthesis tool from Mentor Graphics was used. Release 2011a was used in this project.

- **mike_rt**

  For many exploratory and testing purposes, a multicore-enabled CPU ray-tracer was implemented in full by the author. The application is written in C. Parallel support is accomplished through the pthreads API, and frame buffer operations utilise the Simple Direct Media Layer (SDL) library [SDL13]. Integration with the popular Bullet Physics Engine was implemented also [Bul13]. The ray-tracer is a Whitted-style renderer, supporting shadows, reflection and refraction. Support for kd-trees, uniforms grids and BVHs is included. Importantly, BVH support includes a binned SAH builder which has been specifically tuned to produce identical trees to the proposed hardware. A

Option	Description
-m	*mode* option, used to turn off rendering when only BVH statistics are desired.
-U	Dump all primitives in the order that the upper builder should hand them off to subtree builders in the BVH hardware.
-I	Dump the leaf node indices of the BVH to a file after construction.
-N	Dump the nodes of the BVH to a file after construction.
-b	After loading an obj file into the renderer, dump the primitive AABBs in a format that the hardware understands.
-t	Inform the renderer of the chosen subtree builder capacity. Used in the generation of other statistics.
-n	Inform the renderer of the maximum number of threads supported in the subtree builder. Used in the generation of other statistics.

Table 3.4: *mike_rt command-line options.*

full SAH sweep build BVH was also implemented in mike_rt for SAH quality comparisons [WBS07]. Furthermore, the mike_rt application supports many project-specific operations, accessible via command-line options, that cannot be found in other renderers, and that were used extensively to explore the behaviour of BVH construction under various conditions, and also for design testing. Table 3.4 lists project-specific mike_rt options.[1]

- **Testing**

  Testing of the architecture was achieved through tight integration of the Questasim RTL simulator and the mike_rt renderer via Makefiles. The hand-coded VHDL testbenches support extensive file logging and reporting. Equivalent procedures were implemented in the mike_rt renderer, and diff operations allowed identification of bugs and issues. For more extensive tests, automated regression testing was implemented into these Makefiles which could be run outside of lab hours. Finally, various auxiliary tasks, such as file processing, were implemented with Python and bash scripts.

Summary

In this chapter, the design goals which guided the design and implementation of the BVH construction hardware were presented. Based on these guidelines, a top-level view of the architecture was arrived at, establishing the decomposition of the system into subtree builders and an upper builder. A detailed look at the subtree builder microarchitecture was then presented. Low-level schematics showed the internal microarchitecture of each of the major components of the subtree builder, and a detailed account of their operation was given. Next, this chapter examined the tree output logic, including the storage format of the generated tree, as well as the mechanisms used to generate it. Once the functioning of the subtree builder was established, the issue of system integration was addressed, as well as how the upper builder was derived from the more complex subtree builder microarchitecture. Finally, details on implementation and testing were documented.

In the next chapter, a thorough evaluation of the presented microarchitecture is conducted along a variety of metrics.

[1] mike_rt also supports additional options which are commonly found in other ray-tracers, but they are not listed here as they are numerous and are not of specific interest to the hardware.

Chapter 4

Evaluation

In this chapter, a thorough evaluation of the proposed hardware is conducted. First, the set of test scenes used to evaluate the hardware are presented. Secondly, the parameter space of the subtree builder is explored, showing the behaviour of the builder under a number of conditions. Finally, the subtree builder is combined with the upper builder presented in Chapter 3 to produce a fully fixed-function BVH construction solution. This hardware is evaluated along a variety of metrics, including performance, area efficiency, power efficiency and the quality of the hierarchies produced.

4.1 Test Scenes

To assist with the evaluation, a set of frequently used standard test scenes were collected from a variety of sources. Figure 4.1 shows this collection. Test scenes *Marbles*, *Cloth*, *Fairy Forest* and *Exploding Dragon* were taken from the Utah Animation Repository [Uni13]. The *Conference* model was originally created by Anat Grynberg and Greg Ward, and the *Sponza* model was originally created by Marko Dobrovic, but the author's copies of these scenes were obtained from Jacco Bikker's website [Jac13]. The *Bunny*, *Armadillo*, *Dragon* and *Happy Buddha* scenes were taken from the Stanford 3D Scanning Repository [Sta13]. All models are triangle models.

The collection of models was chosen to represent a wide range of scene complexity. The collection is also purposely designed to incorporate a variety of triangle distributions, including small uniform triangles in models such as *Stanford Armadillo* and *Stanford Dragon*, and long, thin triangles found in architectural scenes such as *Sponza* and *Conference*. All of these scenes are widely used as benchmarks in academic publications on ray-tracing.

4.2 Subtree Builder Analysis

Given the separation by-design of the BVH hardware into subtree builders and upper builders (whether such upper builders represent further fixed-function or not), it will be informative to first examine the subtree builder component in isolation, to determine how performance and efficiency varies across the various possible instantiations. Exploring the design space in this way will ultimately allow the best instantiations for particular scenarios to be determined.

As the subtree builder is not designed to construct hierarchies for large scenes by itself (but only smaller subtrees), it is not possible to directly utilise the test scenes of Section 4.1 to perform this initial

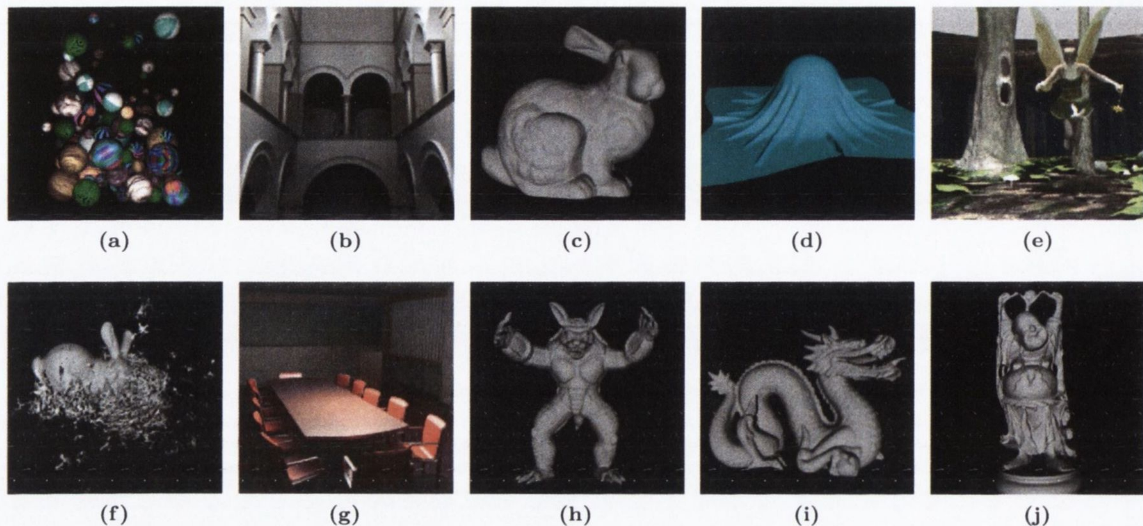


Figure 4.1: *Test scenes used to evaluate the hardware BVH builder: (a) Marbles (8k), (b) Sponza (67k), (c) Bunny (69k), (d) Cloth (92k), (e) Fairy Forest (173k), (f) Exploding Dragon (252k), (g) Conference (282k), (h) Stanford Armadillo (385k), (i) Stanford Dragon (871k) and (j) Happy Buddha (1087k).*

investigation. Rather than generating contrived test cases for the subtree builders, it would seem more prudent to utilise data from real-world test scenes in some form, so as to eliminate the possibility of anomalies arising from artificial or contrived data-sets.

To achieve this, a set of subtrees were extracted from the larger test scenes detailed in Section 4.1 in a specific way. The author's CPU BVH builder was configured to build hierarchies for the larger data-sets, but only down to a large leaf size (8192 primitives for these experiments). As each leaf node naturally arises during the construction of the shallow tree, it represents a realistic build task. The contents of each shallow leaf was exported as a separate OBJ file from the mike_rt application and imported into the hardware simulations. Five subtrees were extracted from the data-sets; one each from Marbles, Cloth, Conference, Stanford Armadillo and Stanford Dragon. These models ranged from 4408 to 7488 primitives in size.

In these experiments, the capacity of the subtree builder was held constant at 8192 primitives. In deciding on the subtree builder capacity for the experiments, a number of factors were considered. In a system utilising the subtree builder to construct larger hierarchies, a larger buffer size should result in faster construction times, as more of the hierarchy can be built on-chip. On the other hand, the buffer sizes should remain within a reasonable hardware cost. In addition to this, the benefit of larger buffer sizes is also dependent on the performance at which the upper levels of the hierarchy can be constructed in any particular solution. Moreover, the size and nature of the input affects this. Smaller scenes or hierarchies being built from a few thousand pre-computed clusters (see Section 3.4) may not benefit from larger buffers, whereas a conventional BVH builder likely would (given its larger working set). In summary, the optimal subtree builder capacity is highly dependent on the intended use of the builder.

In light of these factors, a buffer capacity was chosen that was sufficient to thoroughly exercise the subtree builder, allowing the major trade-offs to emerge, while also being realistic in terms of hardware cost. 8192 primitives results in a total primitive buffer capacity of 432KB per subtree builder, whereas

the L2 cache size of modern high performance CPUs is in the region of 1-2MB.

All results for the subtree builder were measured assuming that the subtree builder was completely unconstrained by memory bandwidth in writing out the hierarchy. The time to fill the subtree builder with primitives was also factored out. This was done to obtain a pure measure of construction performance, as both of these factors could be strongly affected by the performance of other components in any particular use case of the subtree builder component. In any case, a quick analysis shows that the time to fill the subtree builder is 3.6% of the construction time on average (assuming that the maximum filling rate is met), and peaks at 8.8% in these measures. Assuming the maximum filling bandwidth can be met, the filling time is entirely determined by the number of primitives and the number of separate write ports on the primitive buffers.

4.2.1 Performance Analysis

The first major target for profiling is to investigate the variation in construction performance across a variety of instantiations of the subtree builder, and to identify the major factors determining this performance in a variety of use cases. BVHs for each of the five smaller test scenes were constructed using a variety of instantiations, and the execution time (in clock cycles) was measured using RTL simulations.

Recall from Chapter 3 that a subtree builder instantiation is defined by three major parameters:

- ***Width*** (**w**) This determines the number of partitioning units (w), primitive buffers ($2 \times w$) and binning units ($3 \times w$).
- ***Primitive Buffer Capacity*** (**Psize**) The number of primitives each primitive buffer can hold.
- ***Number of Concurrent Threads*** (**Tsize**) This determines the number of active construction paths being followed in the builder at once ($Tsize$), and also the number of SAH calculators present ($2 \times Tsize$).

In addition to these parameters, the runtime parameter l determines the leaf size at which construction will terminate. Although it does not affect the hardware instantiation, we will see that it has a strong effect on the choice of the other variables.

In these experiments, both w and $Tsize$ were varied as powers of two from 2 to 16 inclusive. The total capacity of the subtree builders was held constant at 8192 primitives, and the value of $Psize$ was varied to achieve this value for each instantiation (the number of primitive buffers is controlled by w, and so the size of each buffer $Psize$ must be changed to hold the capacity constant). The leaf size was varied as powers of two from 4 to 64 inclusive, which covers the full spread of leaf sizes commonly found in the literature for various traversal implementations. [1]

Figure 4.2 illustrates the main findings of these measurements as plots showing the relationship between w, $Tsize$ and speedup compared to the simplest possible instantiation ($w = 2$, $Tsize = 2$). The figure consists of four graphs; the top two graphs show the relationship for the smallest leaf sizes (4 and 8), and the bottom two graphs show how this relationship changes for larger leaf sizes (32 and 64).

We first turn our attention to the upper two plots in Figure 4.2. The most salient feature of these plots is that they demonstrate that, for small leaf sizes, the number of active threads (as determined by $Tsize$) is by far the major bottleneck to performance. Increasing the value of w has little effect, unless

[1] Most published implementations utilise leaf sizes of about 2 - 10 triangles per leaf [Wal07, GL10, Ern12]. However, even very large leaf sizes can be beneficial with some traversal methods, as seen for example with vertex culling implementations, which can store dozens or hundreds of triangles per leaf while exhibiting near optimal rendering efficiency [Res07].

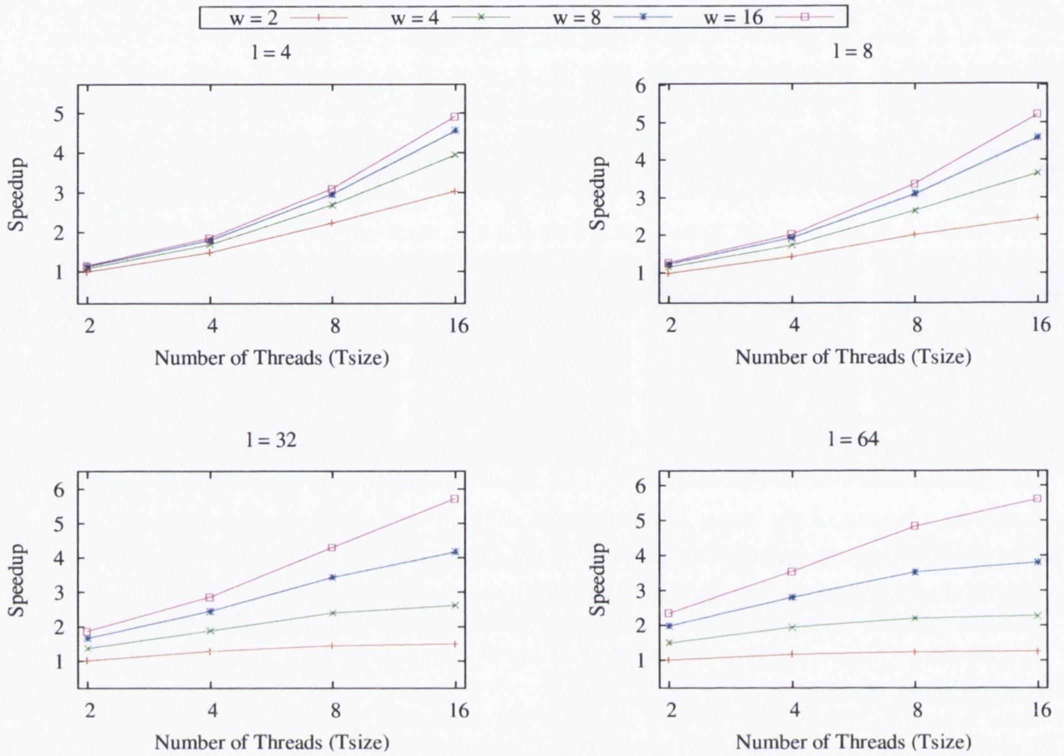


Figure 4.2: *Design space exploration of the subtree builder. Each graph shows the variation in performance of the subtree builder when varying w and $Tsize$ for a different leaf size l. For each graph, w and $Tsize$ are varied from 2 to 16. The top two graphs show the scaling behaviour of the subtree builder for relatively small leaf sizes, whereas the bottom graphs illustrate the behaviour for larger leaf sizes.*

$Tsize$ is also increased. As $Tsize$ grows, the lines begin to diverge more and more (the effect of increasing w itself increases), as can be seen by comparing the $Tsize = 2$ to the $Tsize = 16$ data points in the top two plots.

Moving our attention to the bottom two plots reveals that quite a different picture is obtained for larger leaf sizes. In particular, the $l = 64$ plot on the lower right shows that for small values of w, increasing $Tsize$ has almost no effect on performance. Furthermore, the lines are much more clearly differentiated for all values of w compared to the $l = 4$ case, showing that w is the major bottleneck for large leaves.

The top-down construction method employed by the subtree builder allows us to view the algorithm as a combination of two processes; primitive partitioning and SAH evaluation. Given that w controls the number of partitioning and binning units, it essentially controls the throughput of the partitioning operations in the subtree builder. Similarly, $Tsize$ controls the number of concurrent SAH evaluations, and thus determines the throughput of SAH calculation. The much greater influence of multithreading for lower leaf sizes can be understood by observing the fact that, as the depth of the tree increases, the number of partitioning operations increases linearly (once per primitive per level of the tree), whereas the number of SAH evaluations increases exponentially (as the number of nodes in the tree more than doubles for each new level). Thus, SAH evaluations become much more frequent at smaller leaf sizes. The time

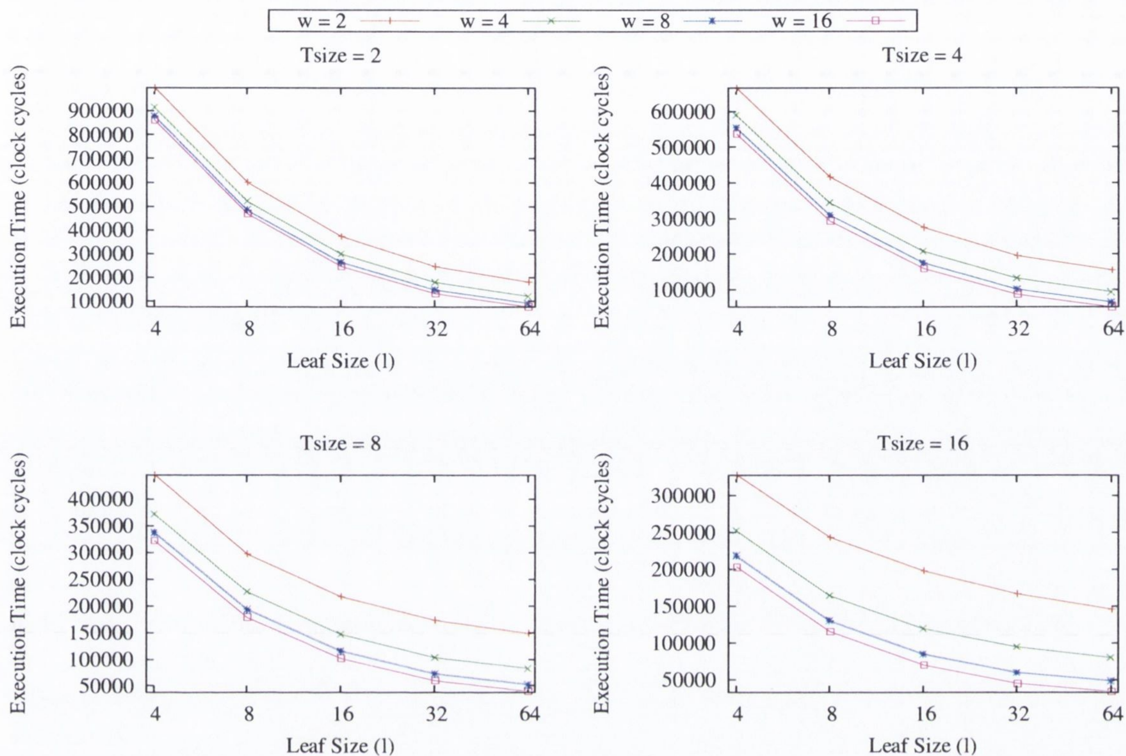


Figure 4.3: *Effect of leaf size on construction performance. Increasing the leaf size always results in faster hierarchy construction, but the degree of speedup is dependent on the particular instantiation of the subtree builder.*

spent waiting on SAH evaluations to complete therefore becomes a larger proportion of the construction time, leading to poor utilisation of the partitioning and binning units if a larger number of threads is not available for them to switch between when they become idle.

In any case, these plots show that the multithreading feature of the subtree builder is effective for the most common range of leaf sizes seen in interactive ray tracers ($l < 10$). For leaf sizes such as these, increasing the number of threads results in as much as a 5× speedup moving from 2 to 16 threads, depending on the value of w and l. In fact, increasing $Tsize$ is the only way to achieve significant speedup for constructing such deep hierarchies, within this range of instantiation parameters. For larger leaf sizes, performance is more sensitive to the value of w, and thus the speed of partitioning operations.

Effect of Leaf Size on Absolute Construction Performance

It has now been established that leaf size l plays a central role in determining the performance bottleneck in the subtree builder. Another important question is how absolute performance is influenced by leaf size. Figure 4.3 documents this behaviour.

Algorithmically, as the leaf size is increased, smaller trees will be produced, resulting in less work being performed. Therefore, we would expect a significant reduction in construction time as leaf size increases. Figure 4.3 shows the sum of the execution time (in clock cycles) of the five smaller testscenes

across the full range of subtree builder instantiations. Each plot corresponds to a different value of $Tsize$, and each plot shows how construction time varies by increasing leaf size for each value of w.

Examining Figure 4.3 confirms that leaf size is a strong determinant of execution time, but reveals that the degree to which execution time is reduced is strongly dependent on the other variables in the system also. If we turn our attention to the top left of the figure ($Tsize = 2$), we notice a large reduction in execution time of approximately one order of magnitude moving from a leaf size of 4 to a leaf size of 64. However, when $Tsize$ is set to 16 in the bottom right graph, the reduction in execution time is only around 2 - 3$\times$. In fact, a consistent downward trend in the speedup achieved is observed as $Tsize$ is increased from graph to graph.

The reason why such a pronounced difference is observed between the two situations lies in the effect of utilisation. Increasing $Tsize$ improves utilisation in the subtree builder by allowing the partitioning and binning units to switch threads when they become idle. This fact was illustrated in Figure 4.2 as increasing w only had an effect when $Tsize$ was also increased.

When the number of threads in the system is small (e.g. $Tsize = 2$), the construction of small nodes in the hierarchy is disproportionally expensive, as a low number of these small jobs (low $Tsize$) is not sufficient to keep the various units busy. As a result, when building down to a small leaf size, more genuine work *and* more stalling are incurred. When construction is terminated at a large leaf size (such as $l = 64$), utilisation is much improved, as large nodes are more capable of keeping the units busy, even when there is only a small number of them active in the system at any one time. In this case, the reduction in execution time is more a result of less genuine work being performed, rather than the avoidance of stalls.

It is clear from these findings, that much attention must be paid to choosing an appropriate number of each type of unit in the design for its desired purpose, and that poor utilisation can arise if the instantiation parameters are not carefully chosen.

Implication for Reconfigurable Systems

In this section, it was demonstrated that the chosen leaf size has a strong effect on the performance of a given instantiation of the subtree builder. It was shown, for example, that increasing the number of SAH calculators is most beneficial when small leaves are desired, and is almost unnecessary when large leaves of 64 primitives or more are called for (unless w is very large). This observation leads to an interesting corollary. Some researchers working in the field of ray-tracing hardware have proposed the use of reconfigurable hardware for implementing more fixed-function aspects of the ray-tracing pipeline [KSS*13, LSL*13]. This offers the advantage that a hardware algorithm need not be entirely fixed at the manufacturing stage, but can be dynamically instantiated in the reconfigurable fabric before or during execution. As mentioned previously in this section, the optimal leaf size can vary widely across different ray-tracing algorithms, generally being below ten primitives per leaf, but rising to several dozen or hundreds for vertex culling implementations. As we have also established that the dominant bottleneck in the subtree builder depends quite heavily on the desired leaf size, it may be beneficial in a reconfigurable system to dynamically configure the most effective instantiation depending on the requirements of whatever traversal method is currently in use.

4.3 Full BVH Builder Analysis

Now that the behaviour of the subtree builder has been established in some detail, we can turn our attention to the ultimate goal of evaluating the performance of the BVH hardware when integrated in a system that is capable of constructing hierarchies for scenes of arbitrary size. As discussed in Section 3.4, there are many possibilities for constructing such a system, and the current experimental platform adopts an *upper builder* approach, where a cut-down, single-threaded builder is used to construct the upper levels of the tree in DRAM.

The RTL code implementing the BVH hardware is highly configurable and a large number of instantiations are possible. For example, the data of Figure 4.2 encompasses 64 configurations of the subtree builder alone (including l). By including the upper builder, we must also factor in the number of RAM pairs and the number of subtree builders themselves, which, within realistic parameters, multiplies the design space by at least another 16×. Given that hardware simulations can take several days, comprehensive coverage of this space for substantial scenes is not feasible with such a detailed simulation model.

As a result of a number of ad-hoc experiments, two instantiations were selected for detailed examination. The first instantiation, which we will call "HWBVH-C" is a compact instantiation that is designed to consume few hardware resources, and thus could be suitable for inclusion in a mobile device. The second instantiation we shall call "HWBVH-HP", a high-performance instantiation which is designed to compete with the performance of highly-optimised software implementations, while also consuming relatively few hardware resources.

As we wish to construct hierarchies that will be useful to a typical interactive ray-tracer, both instantiations are configured to construct deep trees with a leaf size of 4, which is a typical value for many software ray-tracers [1] . As was established in Section 4.2.1, the chosen leaf size strongly influences the efficiency of a given subtree builder instantiation, with the number of threads ($Tsize$) being much more important for overall performance for small leaf sizes. This information was used to inform the selection of subtree builder for these experiments. Both HWBVH-C and HWBVH-HP utilise the same subtree builder configuration. In each case, the subtree builders always use 16 bins (the maximum currently supported in the hardware) and always check all three axes for SAH evaluations. Table 4.1 details the full set of parameters.

Instance	Clock Freq.	# of RAM pairs	# of Subtree Builders	w	$Tsize$
HWBVH-C	250 MHz	1	1	4	8
HWBVH-HP	500 MHz	2	2	4	8

Table 4.1: *Tested HWBVH Instantiations.*

For clock speeds, very conservative values are assumed for these simulations. Both values are similar to those assumed in other works on ray-tracing architecture [SBK*08, NPP*11]. The faster speed of 500 MHz is only one-third to one-half the clock frequency of a typical manycore platform. For example, the NVIDIA GTX 580, released in 2010, clocks its shader cores at 1544MHz, and the recent Intel Xeon Phi Coprocessor 7120D runs its 61 cores at 1238MHz, and can turbo boost to as high as 1333MHz. For CPUs, the AMD FX-9590 runs at 4700 MHz; a clock rate that is almost a full order of magnitude faster than is assumed here.

[1] For instance, the default parameters for the bvh2 data structure in Intel Embree produces leaves of approximately this size.

To model external memory, the DRAM model described in Section 3.5.3 was utilised. Each memory channel in a RAM pair at peak is capable of delivering one 192-bit word per cycle, and runs at the same clock frequency as its attached BVH hardware instantiation. The highest bandwidth delivered is therefore to the HWBVH-HP instantiation, at around 44GB/s max. Again, this is very conservative, as memory systems on modern manycore chips often deliver in excess of 200GB/s of bandwidth.

The BVH hardware is intended to reside on-chip as a device on the system bus, rather than an off-chip co-processor. For this reason, no communication with a host CPU or GPU is included in the execution times.

4.3.1 Performance Analysis

To evaluate the construction performance of the BVH hardware, both instantiations were compared to four other published BVH builders. Each builder is the highest-performing implementation of its kind to the author's knowledge. All builders are specifically optimised for ray-tracing applications.

The first two builders represent the state-of-the-art in top-down binned SAH BVH construction, with an Intel MIC implementation by Wald [Wal12], and the highly-optimised *bvh2* binned SAH implementation available in the Intel Embree set of ray-tracing kernels [Ern12]. The Embree code was executed on the most powerful machine available to the author, a dual CPU Intel Xeon E5-2620 system featuring 12 cores running at 2 GHz and 32 GB of RAM. During execution, examining *htop* confirmed that Embree was using all 24 hardware threads on this machine.

The second two builders are more approximate hybrid-style builders; the first being the fastest published implementation of HLBVH [GPM11], and the second being the refinement-based TRBVH [KA13]. In the case of TRBVH, results are shown for the "non-splitting" version of the builder, which is the fastest version of this algorithm.

As simulating the BVH hardware in the Questasim RTL simulator was extremely time-consuming (the larger scenes took several days to simulate), it was infeasible to build each frame of the animated test scenes used in this evaluation (Marbles, Cloth, Fairy Forest and Exploding Dragon). For these scenes, the middle keyframe was used. Figure 4.4 summarises the performance results. Note that data for all scenes was not available for each implementation, as different publications use slightly different test scenes.

Taking first the HWBVH-C instantiation, it is evident from the figure that this instantiation is capable of constructing hierarchies in some cases more quickly than the Intel MIC implementation and also Embree running on the dual Xeon system. Only for the two largest scenes (Dragon and Buddha) is HWBVH-C significantly slower than these platforms. As HWBVH-C is designed primarily for efficiency, this can be considered a positive result. Section 4.3.4 completes this picture by comparing these systems in terms of area and power efficiency.

By examining the performance figures for HWBVH-HP, it is clear that this instantiation far exceeds the performance of state-of-the-art binned builders executing on high performance manycore and multicore architectures by a factor of between 3 and 10×. Comparing HWBVH-HP to the TRBVH executing on a GTX Titan shows that it can approximately match the performance of this implementation, being slightly slower only on Buddha and actually faster on some of the other scenes. This is significant, as the TRBVH represents a less costly algorithm, and is executing on one of the most powerful GPUs ever released at the time of writing. Once again, the full picture of how these implementations compare will become apparent when power and area efficiency are taken into account.

Overall, the HLBVH implementation delivers the fastest construction times in most cases. However,

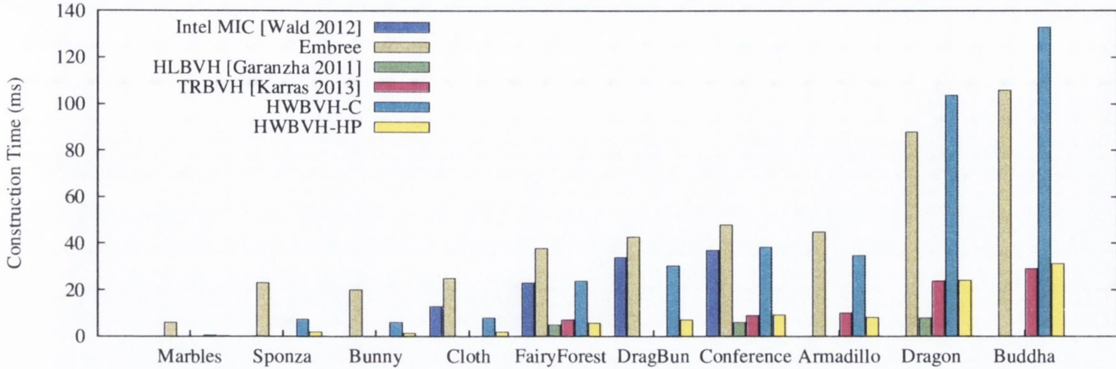


Figure 4.4: *Construction performance of the BVH hardware vs. competing software BVH builders.*

comparing HLBVH directly to the binned builders shown in this comparison, and even TRBVH, hides the fact that HLBVH represents a vastly different point in the tradeoff between construction performance and traversal efficiency. As a result of this, HLBVH usually delivers considerably lower quality hierarchies. The quality of HLBVH hierarchies is further elaborated on in Section 4.3.3. Even so, the BVH hardware is actually capable of constructing hierarchies for some scenes in fewer clock cycles than HLBVH, extrapolating from the construction times and clock frequency of the GPU.

Overall, we observe that our implementation can deliver high quality, high performance binned SAH builds at speeds faster than current multicore and manycore implementations. This high performance is achieved through low-latency/high-bandwidth primitive buffers delivering very efficient streamed data access to the rest of the circuit, which consists of a set of very fast dedicated units for the expensive SAH evaluation and binning.

4.3.2 Memory Behaviour

Another important consideration for the BVH hardware is the overall memory behaviour that it exhibits. There are a number of important aspects of memory behaviour, but three are of particular importance; the total memory bandwidth consumed (i.e. total memory traffic generated), the total footprint of temporary data used during construction, and the memory footprint of the output hierarchy itself. Furthermore, measuring the relative contribution of each type of data element to these figures will be useful, as it may inform future optimisations.

Bandwidth Utilisation

We first turn our attention to the total memory bandwidth consumed during construction. Memory traffic in the BVH hardware can be broken down into four sources: 1) the data read from memory for calculating the initial scene AABB; 2) the data read and written by the upper builder (AABBs and indices) for constructing the upper levels of the hierarchy (we shall call this "Upper Sort"); 3) the bandwidth consumed in writing out hierarchy nodes; and 4) the bandwidth consumed in writing out hierarchy indices.

Regardless of the parameters used to instantiate the hardware, the data read to calculate the initial scene AABB and the size of the output hierarchy will be the same. The only other source of memory

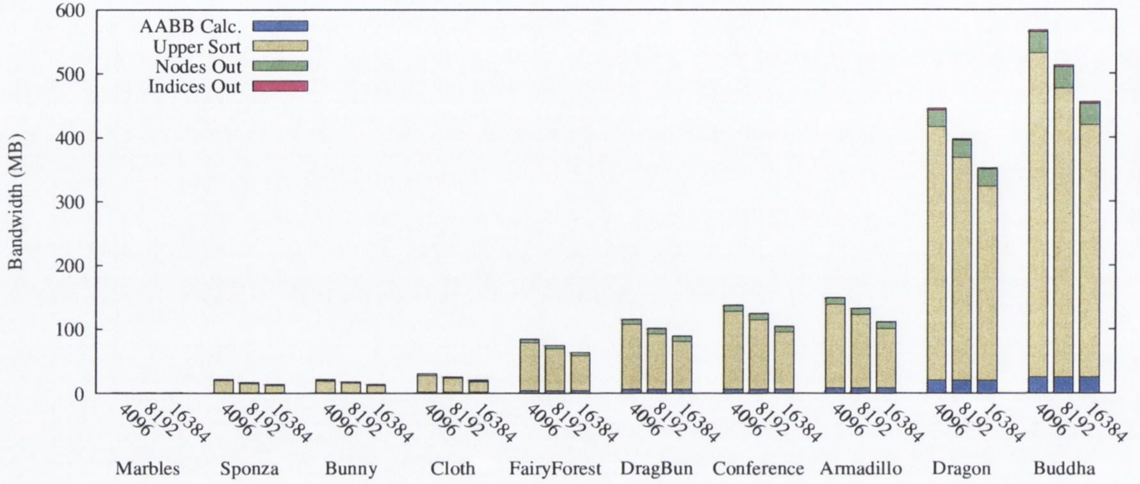


Figure 4.5: *Breakdown of the memory traffic produced by the BVH hardware. Total memory traffic increases reliably with the scene size (scene size is ascending from left to right). Each cluster of bars shows the memory traffic generated using a different primitive buffer capacity for the subtree builders in the system. It is clear that the memory traffic generated by the upper builder in constructing the upper nodes ("Upper Sort") dominates the memory traffic, and becomes an increasingly larger portion of the traffic as the scene size increases.*

Marbles	Sponza	Bunny	Fairy Forest	Exp. Dragon	Conf.	Arma.	Dragon	Buddha
0.45	3.47	3.57	8.89	13.01	14.56	17.82	44.90	56.02

Table 4.2: *Memory footprint (in MB) of hierarchy construction for each of the test scenes.*

traffic is the "Upper Sort", and this will depend on the size of the primitive buffers in the subtree builders in any given instantiation of the hardware. Therefore, HWBVH-C and HWBVH-HP will generate the same total amount of memory traffic over hierarchy construction, as their subtree builder capacities are the same. Increasing the size of these buffers will allow the upper builder to hand off subtrees earlier to the subtree builders, thus reducing the traffic generated in the "Upper Sort" phase. To investigate the effect of changing primitive buffer capacity on total external memory traffic, each scene was constructed using subtree builder capacities of 4096, 8192 and 16384 primitives. Figure 4.5 shows the breakdown of the total memory traffic consumed during hierarchy construction.

A number of important observations can be made from Figure 4.5. As would be expected, the total amount of memory traffic consumed increases as the scene size increases. The AABB calculation and tree output consume only a small fraction of the total bandwidth, whereas the Upper Sort dominates memory traffic in all cases. Moreover, increasing the size of the subtree builder capacity has only a modest effect on the total bandwidth consumed. This modest gain can be explained by the fact that, since nodes in any given level of the tree will encompass approximately half the number of primitives as those nodes one level up in the tree, doubling the subtree builder capacity should transfer about one level of hierarchy construction from the upper builders to the subtree builders. The upper builder will usually build several

levels of the tree for the larger scenes shown here (around 6 or 7), so the reduction is not dramatic.

Bandwidth figures are typically not given in hierarchy construction papers, and the only figures available for comparison were to be found in the HLBVH implementation by Pantaleoni and Luebke [PL10]. Referring to these numbers shows that the BVH hardware consumes about 2-3× less total bandwidth compared to this implementation. Such bandwidth savings could be an advantage when running other tasks in parallel with the builder such as concurrent rendering/hierarchy construction.

Memory Footprint

A second important aspect of memory behaviour is the memory footprint required by the construction process. These values are insensitive to all instantiation parameters of the subtree builder. In the case that multiple memory channels are used, the footprint will be evenly spread over the different memories. Table 4.2 lists the memory footprint for each test scene in megabytes. This footprint does not include the tree output, but only temporary data.

The memory footprint consists entirely of two copies each of the primitive AABBs and the list of primitive indices. As there is one AABB and one index per primitive, each consumes a fixed portion of the memory footprint, regardless of the scene. If 32-bit floating point AABBs and 24-bit indices are used (as is the case here), then 89% of the footprint consists of the AABBs.

4.3.3 Tree Quality

In ray-tracing applications, overall performance is determined not only by the speed of spatial index construction, but also on the efficiency of rendering. While many factors affect the efficiency of ray traversal and intersection in such applications, the efficacy of the data structure at culling superfluous intersection tests is one of the most dominant. Ultimately, the goal in an interactive ray-tracer is to minimise the *time to image*:

$$\textit{time to image} = \textit{spatial index construction time} + \textit{rendering time} \tag{4.1}$$

The accepted procedure for assessing the quality of any new construction approach is to compare its expected traversal efficiency with that of an established "gold standard" reference builder. Easily the most widely used method of estimating the efficiency of ray-tracing data structures is by using the SAH. Just as the SAH can be used for estimating the cost of splitting a node, it can also be used for estimating the quality of the entire hierarchy. A number of similar formulae have been devised for assessing the ray traversal efficiency of a given hierarchy. Most of these have been based on the SAH. In these measures, the recently proposed recursive formula for the SAH cost C was used [PL10]: [1]

[1] An alternative formula which gives very similar results was presented by Wald et al. [WBS07].

	[Wal12]	[Ern12]	[PL10]	[KA13]	HWBVH
	Binned SAH	**Binned SAH**	**SAH HLBVH**	**TRBVH**	**Binned SAH**
Marbles	-	106%	-	-	101%
Sponza	-	98%	-	-	103%
Bunny	-	103%	-	-	101%
Cloth	-	103%	-	-	101%
Fairy Forest	100%	100%	-	99%	103%
DragBun	103%	117%	-	-	105%
Conference	101%	97%	-	83%	114%
Armadillo	-	108%	109%	106%	100%
Dragon	-	101%	112%	107%	101%
Buddha	-	103%	114%	106%	102%
Mean	102%	103%	112%	101%	103%
Median	101%	103%	112%	106%	102%
Min	100%	97%	109%	83%	100%
Max	103%	117%	114%	107%	114%

Table 4.3: *Comparison of the SAH costs produced by the BVH hardware with those of competing implementations. Tree costs for HLBVH are taken from the original HLBVH by Pantaleoni and Luebke [PL10]. Although triangle splitting can be enabled in TRBVH, quality results are shown for the "non-splitting" version, which is the version used for the performance estimates in Section 4.3.1. Actual SAH costs and node statistics for the BVH hardware and for the author's sweep builder are included in Table 4.4.*

$$
\begin{aligned}
C(node) &= T_1 \cdot N_{children} + \sum_{i}^{N_{children}} C(child_i) \cdot P_{hit}(child_i) \\
C(leaf) &= T_2 \cdot N_{children} \\
P_{hit}(node) &= \frac{area(node)}{area(parent)}
\end{aligned}
$$

where $T1$ and $T2$ are the cost of intersecting a bounding box and a scene primitive respectively, and $P_{hit}(node)$ is the probability of intersecting a node assuming that a given ray has intersected its parent. Following the original authors, $T1$ and $T2$ are both set to 1 in these measurements.

Since both BVH hardware instantiations use the same binning method and termination criterion, they produce identical hierarchies. To establish the quality of the trees produced by the BVH hardware, their *SAH* costs were measured using the equation above. [2] A full SAH sweep builder was also implemented in mike_rt as the gold standard [WBS07], and supports full SAH-driven leaf node termination. By definition, the SAH cost of the sweep builder is taken as 100%, and all costs are expressed relative to the sweep builder cost. In these measurements, the BVH hardware samples all three axes with 16 bins, and terminates at a fixed 4 triangles per leaf. In this comparison, it is SAH *cost* that is measured, and so lower values are better. Table 4.3 summarises the results.

A number of important observations are to be made in this data. Firstly, it is evident from the data that the BVH hardware produces quite similar SAH costs compared to the two state-of-the-art binned SAH implementations, with similar mean and median values. Furthermore, the table replicates the known

[2] To obtain more comparable cost estimates, this equation was also implemented inside Embree as an additional method in the "bvh2" class. Embree's own quality metric differs somewhat from the more commonly used metrics.

	BVH Hardware	# nodes	Sweep Builder	# nodes
Marbles	125.6	5,827	124.2	6,009
Sponza	118.4	43,057	115.7	42,903
Bunny	57.85	42,469	57.24	49,157
Cloth	46.52	59,501	46.0	63,445
Fairy Forest	50.43	112,059	49.0	106,387
Drag. Bun.	32.8	168,407	31.4	175,553
Conference	79.0	185,221	69.4	177,255
Armadillo	52.8	208,089	52.6	253,565
Dragon	90.2	575,971	88.8	575,571
Buddha	102.0	719,903	100.6	715,635

Table 4.4: *SAH costs and number of hierarchy nodes for both the BVH hardware and the reference sweep builder, with KI and KT set to 1.*

result that binned SAH algorithms usually give SAH costs very close to full sweep builders. For the BVH hardware, an exception to this can be seen with the Conference scene (SAH cost of 114%), which is not surprising as other researchers have noted difficulty with binned builders in this scene [Wal07, LGS*09]. This is likely the result of large triangles in this scene causing a higher degree of node overlap in the hierarchy.

Compared to the HLBVH implementation, we see significantly higher quality across the board for the binned SAH builders, including the BVH hardware. This is expected, as the HLBVH performs SAH-driven splitting on only a small fraction of the hierarchy nodes, with the rest of the hierarchy effectively split on the spatial median. Comparing to TRBVH shows that this algorithm performs quite well on scenes featuring large triangles (Fairy Forest and Conference) when compared to the binned builders, but seems to deliver slightly lower quality hierarchies for scenes with evenly-sized geometry, such as on the scanned models like Armadillo and Dragon.

In summary, the BVH hardware delivers hierarchies of competitive quality to many other interactive builders, and approximately matches existing binned SAH implementations. Moreover, there are many possible optimisations for improving tree quality in future versions of the hardware. Firstly, increasing the number of bins utilised in the upper builder would result in only a negligible increase in hardware cost and construction time, but could deliver significant quality improvements. Increasing the number of bins utilised in the subtree builders would be more costly, both in terms of hardware and construction performance, but mitigating this cost would be a potentially useful avenue for further investigation. Additionally, the BVH hardware is compatible with various triangle pre-splitting schemes, such as the technique of Ernst and Greiner [EG07], which is likely to significantly improve hierarchy quality for scenes with uneven triangle sizes such as Conference.

One final point which should be acknowledged is that recent work has shown that the SAH is a somewhat imperfect predictor of the ray-tracing performance of BVHs [AKL13]. However, although imperfect, this work demonstrates that the SAH provides at least a rough estimate of practical ray-tracing performance. Specifically, the SAH achieves an average correlation of 0.915 with practical performance over 22 test scenes in this paper, which rises to 0.933 on SIMD architectures. Furthermore, it has been established in other works that the binned approach does indeed yield practical performance very similar to full sweep builds when the build parameters are similar to those used in this thesis [Wal07, LGS*09]. Therefore, the author considers these results to be a fair representation of the efficiency of the hierarchies produced by the BVH hardware.

4.3.4 Area and Power

It is envisaged that the BVH hardware could be useful to a heterogeneous graphics processor. Since including the unit in such a processor would consume additional hardware real-estate, it is important to estimate the area cost of including it in such a system. Furthermore, as was argued in Chapter 1, power consumption has become one of the most important measures of computing efficiency. It is thus important to estimate the degree to which the BVH hardware could contribute to power efficiency in a hardware rendering system.

To perform such an analysis, the free version of the Cadence InCyte Chip Estimator was used to estimate area and power based on a variety of macro-level statistics, including the number and type of hardware intellectual property (IP) needed (floating-point units, register space, SRAM, etc.), the expected clock frequency, and the desired technology node.

Estimation Procedure

The first step in this estimation process is to tabulate the number and type of the major hardware IP components in the design. Table 4.5 shows the required number of floating-point cores, register space and SRAM needed for each major design unit in the subtree builder. Similarly, Tables 4.6 and 4.7 show similar tabulations for the upper builders featured in HWBVH-C and HWBVH-HP respectively. Table 4.8 shows the overall totals of the two builder instantiations. These tables in themselves represent a technology-generic expression of required resources.

Based on these tabulations, a separate InCyte Chip Estimator project was created for the HWBVH-C and HWBVH-HP. InCyte allows estimations to be performed for entire chips, but breaks down resources into a number of components, including the *core logic*, which will allow us to examine the area and power of the BVH builder hardware in isolation.

On startup, InCyte launches the Quick Estimation wizard, which collects the most important attributes of the design for estimation, including expected clock frequency, number and type of the IP cores used, and the technology node desired. Using these tabulations, IP cores were selected from the InCyte library which most closely matched the IP utilised in the design.

The free version of InCyte used in these estimates is feature-limited. In particular, the lowest technology node which can be used in estimations is 65nm. To perform estimates, InCyte provides a library of IP with each technology node, with some IP only available in certain nodes. By careful examination of these libraries, it was found that the 90nm IP library provided IP that was most similar to the IP utilised in the BVH hardware. Therefore, estimates were performed for the 90nm technology node.

InCyte's fundamental estimation approach is based on choosing IP from its internal libraries which most closely match the IP used in the design. To match the floating-point cores featured in the BVH hardware, the corresponding FP units from the Digital Core Design IP library in InCyte were chosen, as these closely matched the parameters of the cores utilised in the BVH hardware. As a floating-point comparator is not featured in this library, the comparator utilised in the BVH hardware was synthesised to an ASIC standard cell library in the Mentor Graphics Leonardo Spectrum tool. InCyte allows specifying custom IP by gate count, and so the results of this synthesis were used to define the comparator in InCyte, as consisting of 235 gates.

Given the large size of the primitive buffers, it is likely these would be implemented in SRAM. Therefore, we must add 432KB of SRAM for each subtree builder in the tabulations.

The final value to be delivered to InCyte is the number of miscellaneous gates in the design which

	Part. Unit	Bin. Unit	SAH Calc.	Tree Output	Design Total
# Used	4	4	16	1	
FP ADD	1	3	9	0	160
FP SUB	3	9	9	0	192
FP MUL	1	3	12	0	208
FP DIV	1	3	0	0	16
FP CMP	0	0	144	0	2304
FXD-FLT	0	0	3	0	48
FLT-FXD	1	3	0	0	16
REG	2KB	4KB	9KB	256KB	422KB

Table 4.5: *Total number of floating point cores and register space needed to implement the subtree builder.*

	Part. Unit	Bin. Unit	SAH Calc.	Tree Output	Design Total
# Used	1	1	2	1	
FP ADD	1	3	9	0	22
FP SUB	3	9	9	0	30
FP MUL	1	3	12	0	28
FP DIV	1	3	0	0	4
FP CMP	6	0	36	0	78
FXD-FLT	0	0	3	0	6
FLT-FXD	1	3	0	0	4
REG	147KB	4KB	6KB	4KB	164KB

Table 4.6: *Total number of floating point cores and register space needed to implement the HWBVH-C upper builder.*

	Part. Unit	Bin. Unit	SAH Calc.	Tree Output	Design Total
# Used	2	2	2	1	
FP ADD	1	3	9	0	26
FP SUB	3	9	9	0	42
FP MUL	1	3	12	0	32
FP DIV	1	3	0	0	8
FP CMP	12	0	72	0	168
FXD-FLT	0	0	3	0	6
FLT-FXD	1	3	0	0	8
REG	147KB	4KB	7KB	4KB	316KB

Table 4.7: *Total number of floating point cores and register space needed to implement the HWBVH-HP upper builder.*

	HWBVH-C	HWBVH-HP
FP ADD	182	346
FP SUB	222	426
FP MUL	236	448
FP DIV	20	40
FP CMP	2382	4776
FXD-FLT	54	102
FLT-FXD	20	40
REG (KB)	587	1161
SRAM (KB)	432	864

Table 4.8: *System totals for HWBVH-C and HWBVH-HP.*

	HWBVH-C	HWBVH-HP
Technology Node	090 nm	090 nm
Random Logic	2,480,000 gates	4,850,000 gates
Clock Speed	250 MHz	500 MHz
Number of SRAM bits	3,538,944	7,077,888
Number of RF bits	4,808,704	9,510,912

Table 4.9: *Values chosen for the InCyte chip estimator options. All other values in the tool were left at defaults.*

	HWBVH-C	HWBVH-HP
Die Area	55.947 mm^2	105.787 mm^2
Die Dimensions	7.480 mm× 7.480 mm	10.285 mm× 10.285 mm
Total Equivalent Gates	13,479,356 gates	26,359,541 gates
Active Power	5.902 W	25.179 W
Standby Leakage	120.312 mW	218.808 mW
Total Power	6.023 W	25.398 W
Technology	90nm/Generic Foundry	90nm/Generic Foundry

Table 4.10: *InCyte Chip Estimator area and power estimates for HWBVH-C and HWBVH-HP.*

are not covered by floating-point cores, register space or SRAM. To account for these, miscellaneous logic was added to consume 20% of the core area in the final InCyte estimations. This value is borrowed from the work of Nah et al. [NPP*11], who use a similar value for estimating the overhead of miscellaneous logic in their traversal engine. In fact, the estimation procedure presented here overall is not dissimilar from that of Nah et al.

Table 4.9 gives the full list of parameters given to InCyte for the two BVH hardware instantiations. All other values were left at defaults.

Estimation Results

Table 4.10 shows the resulting datasheets for both instantiations of the hardware, reproduced directly from the InCyte software. Even in 90 nm, it is clear that both instantiations of the BVH consume relatively little hardware. Comparing to a recent manycore chip, the NVIDIA GTX Titan (as was used in the TRBVH implementation in the performance comparisons of Section 4.3.1) utilises a 561 mm^2 die size at 28 nm, and die photos suggest that the vast majority of this space is spent on the programmable shader cores (i.e. the resources contributing to the performance of BVH construction on that platform). The values of 55.947 mm^2 and 105.787 mm^2 for the die area estimations are significant, especially since they are measured using only 90 nm libraries. We can thus conclude that the hardware cost of the BVH hardware would be minimal if appropriately implemented in a modern technology, such as 28 nm.

Also interesting are the figures generated for Total Power. Since the thermal design power (TDP) of many modern processors is over 100 W in some cases, these figures are quite promising. However, rather than simply comparing to the *TDP*, the author sought more accurate estimates for power consumption of competing implementations.

To achieve this, the Intel Embree ray-tracer was instrumented with performance monitoring code to sample the running average power limit (RAPL) counters available on modern CPUs. This code was executed on two platforms: the first was the dual CPU Xeon E5-2620 system introduced in Section 4.3.1, and the second was a Core i5 3210M laptop processor. The Xeon system represents a high-end platform

	HWBVH-C	HWBVH-HP	Embree Core i5 3210M	Embree 2×Xeon E5-2620
Marbles	0.0040	0.0041	0.03	0.11
Sponza	0.0437	0.0510	0.23	0.47
Bunny	0.0356	0.0367	0.23	0.46
Cloth	0.0482	0.0495	0.31	0.61
Fairy Forest	0.1434	0.1424	0.62	1.17
DragonBun	0.1841	0.1815	0.87	1.77
Conference	0.2320	0.2349	0.98	1.94
Armadillo	0.2101	0.2082	1.25	2.38
Dragon	0.6243	0.6170	3.03	5.62
Buddha	0.8004	0.7986	3.83	7.43

Table 4.11: *Total energy expended (in Joules) during hierarchy construction by the core logic of two BVH hardware instantiations, and by the Embree running on both a mobile Core i5 processor and a dual-CPU server-class Xeon system.*

	HWBVH-C	HWBVH-HP	Embree Core i5 3210M	Embree 2× Xeon E5-2620
Power Consumption	6 W	25 W	10 - 13 W	15 - 70 W

Table 4.12: *Total power consumed (in Watts) by each of the measured BVH construction implementations. Power readings for instantiations of the BVH hardware are taken directly from the InCyte tool.*

that is presumably optimised for performance, whereas the laptop processor is very likely optimised for power efficiency. Therefore, examining both systems should give an idea of the spread of power efficiency seen in modern multicore processors.

The RAPL counters allow for measuring the total accumulated energy (in Joules) over the course of BVH construction. As Embree already reports the time to construct the data structure, we can easily obtain power readings in Watts from the energy readings in Joules. Conversely, since InCyte reports only power in Watts, we can obtain Joules by factoring in the construction times for the various test scenes.

The RAPL counters allow for measuring different components of the processor power, including *core*, *uncore* and (in the case of the server-class Xeon system) *memory power*. As the InCyte estimates for the BVH hardware include only the core logic (and not memory power, caches etc.) the values measured with the RAPL counters are for core power only. In the case of the dual CPU Xeon system, the energy readings are the sum of the cores on both CPUs. Table 4.11 shows the energy expenditure of these platforms while constructing each test scene.

Since the HWBVH-C instantiation is designed to achieve high power and area efficiency (rather than maximum performance) it is most appropriate to compare this instantiation to the mobile Core i5 processor in terms of power efficiency. Taking these two first in Table 4.11, it can be seen that the estimated energy expenditure of HWBVH-C in constructing the hierarchy compares very favourably to the measured energy expenditure of the Core i5. The difference is highly significant, being about a factor of 4-5× less energy expended. Moreover, by examining the corresponding entries for power in Table 4.12, we can see how these energy readings are articulated in power itself. In this case, the total power consumption for HWBVH-C is around half that of the Core i5. However, these readings must be placed in the context of performance also, as the HWBVH-C instantiation is capable of constructing hierarchies about one order of magnitude faster than the Core i5 mobile processor, while still drawing only half the core power to achieve this.

Comparing HWBVH-HP to the dual Xeon system in terms of energy expenditure shows that HWBVH-HP consumes around one order of magnitude less energy than the dual Xeon system. Comparing the HWBVH-HP to the Xeon system in terms of power is a little more complicated. The Xeon system shows significant variation in the power consumption across scenes. Power rises as a function of increasing scene size, so smaller scenes show lower values, and larger scenes consume more power.

This fact can be explained by the variation is scalability of the binned SAH construction with increasing scene size. Small scenes cannot sufficiently engage a large number of cores, and so much of the processor is idle during their construction. This fact was verified by varying the number of threads during construction for all scenes, and recording the execution times. This also explains why such little variation is seen on the mobile Core i5, as this processor consists of only two cores. In the case of larger scenes, much better utilisation and speedup is seen over 12 cores on the Xeon system. Even using very large scenes, such as the 4.3 million triangle Crown scene, did not increase the power much beyond the 70 W recorded for the largest scenes shown here. In practical applications, it is likely that scenes will be large enough to engage a large number of cores (like the Dragon and Buddha scenes tested here). In these situations, HWBVH-HP consumes around 2 - 3× less power than the Xeon system, while delivering around 3× faster hierarchy construction. It can thus be concluded that HWBVH-HP is substantially more power-efficient than the dual Xeon system.

Comparison with Manycore Systems

One question which was not examined in detail is how power efficiency of the BVH hardware compares to manycore platforms, such as a typical GPU. However, we can get some sense of how it compares by examining the reported TDP of typical GPU chips. In these experiments, the measured power of the multicore platforms was approximately half their stated TDP (35W for the Core i5 3210M and 95W for the E5-2620). Taking a specific example for manycore systems, the stated TDP for the GTX Titan GPU (on which the TRBVH implementation used in the comparisons of Section 4.3.1 executes) is 250 W. If one makes the assumption that even a quarter of this will be expended by the core processor logic during BVH construction, the comparison is still highly favourable for the BVH hardware.

Another significant fact which should be restated is that the power measurements made in the InCyte chip estimator were necessarily made in outdated 90nm technology (due to the limitations of the free version). Furthermore, it is highly likely that all of the commercially available processors featured in this section incorporate a significant degree of power-specific optimisations (such as clock and power gating), whereas the BVH does not yet incorporate such techniques, but certainly could do so. Therefore, these estimates can be taken as highly conservative.

Summary

In this chapter, the subtree builder was profiled to explore its behaviour under a number of different instantiations and leaf sizes. It was shown that different instantiations are more appropriate under different circumstances; in particular, the desired leaf size was shown to strongly determine the performance of a given instantiation.

Following this, two full BVH construction solutions were then presented, HWBVH-C and HWBVH-HP, both utilising a fixed-function solution to constructing the upper levels of the hierarchy, and featuring a different number of subtree builders. The instantiations were shown to deliver high-quality hierarchies

at levels of performance far exceeding the state-of-the-art in top-down binned SAH BVH construction. Power efficiency was assessed for both instantiations and found to compare very favourably with two modern multicore processors running highly-optimised BVH builders.

The next, and final chapter brings the thesis to a close by placing these findings in the context of the original research question.

Chapter 5

Conclusion and Future Work

In this chapter, the veracity of the original research statement laid out in Chapter 1 is assessed in light of the empirical findings of Chapter 4. Following this, several avenues for future work are discussed.

5.1 Conclusion

At the beginning of this thesis, the research question was raised as to whether a custom microarchitecture for the construction of BVHs for ray-tracing could offer significant benefits to an interactive rendering system in terms of performance and efficiency.

To test this question, the first ever published microarchitecture for achieving such a task was presented in detail. The design was implemented in full as a VHDL model, and was evaluated along a variety of metrics. The design was compared to state-of-the-art BVH construction algorithms running on modern multicore and manycore platforms, and was found to exceed the capabilities of these implementations by a significant margin in a number of important ways, including raw performance improvements of up to 10×, while consuming very few hardware resources and exhibiting high power efficiency of up to one order of magnitude less energy expended.

These findings are of significance to future graphics processors. The rise of interactive ray-tracing, the dawn of the mobile era, and the emergence of power efficiency as the principal limit to processor scalability, all provide fresh motivation to incorporate fixed-function devices into future heterogeneous processors in order to reap the substantial benefits that they can deliver. It is the author's opinion that, for the domain of graphics processors, and especially for ray-tracing processors, acceleration data structure construction is a strong contender for one such subsystem for which fixed-function designs could be useful.

5.2 Future Work

Given the considerable variety of techniques employed in spatial index construction for ray-tracing, and the wide applicability of BVH-like structures in computer graphics and interactive simulation, a number of avenues for future work naturally present themselves.

The first area of interest would be to see if the BVH hardware could be adapted to also construct other types of acceleration structure. The most obvious candidate for this investigation would be the kd-tree, as it is quite a similar data structure and, importantly, would be of considerable practical utility. Spatial index structures can be combined very flexibly within a single scene, with some structures being

more appropriate in specific circumstances. For example, the consensus in the ray-tracing community is that kd-trees still represent the best option for static or almost static geometry. The largest fundamental difference between constructing a kd-tree as compared to a BVH is that a kd-tree necessitates strict spatial splitting, which most often results in some primitives being referenced in multiple hierarchy nodes. Since the growth of the number of primitive references is not entirely predictable, dynamic memory management would need to be employed to address this. This requirement is incompatible with the current subtree builder design, as it relies on only a fixed, predetermined number of primitives being present in the partitioning buffers. The biggest challenge would therefore be to handle any such "overflow" of the primitive buffers into main memory.

A second avenue for future work would be to begin to integrate the design into a full system comprising programmable components, other fixed-function components, and perhaps some form of reconfigurable hardware. Good candidates for such a host system already exist. In particular, the SGRT architecture [LSL*13] would be ideal, as it includes fixed-function BVH traversal hardware, reconfigurable hardware shaders, and a multicore ARM processor. If the BVH hardware was incorporated into a full system in this way, it would then be possible to implement many of the techniques discussed in Section 3.4, such as combinations of BVH updating and construction, acceleration of hybrid SAH/(H)LBVH builds, and lazy construction strategies.

Another area of future work would involve the implementation of several architectural and circuit optimisations which are already known to the author and are expected to have a considerable positive impact on system performance and efficiency. For example, the SAH Calculator unit (Section 3.3.4) could be broken into two smaller components (one unit to implement accumulation of the SAH bins, and a second to evaluate each bin) which would allow pipelining the two stages, rather than the first stage remaining idle until the second has completed. A further example of a useful optimisation would be to implement more balanced thread creation in the subtree builders (Section 3.3.1), as it sometimes occurs that some threads persist significantly longer than others, which leads to sub-optimal hardware utilisation since an insufficient number of threads is available to hide the latency of the SAH calculation. Related to these optimisations is the broader fact that no *specific* circuit design techniques relating to power efficiency have been utilised in the BVH hardware as of the time of writing. Given the importance of power efficiency in modern processor design, it is highly likely that the competing systems that were compared to the BVH hardware in Chapter 4 have all been considerably power-optimised. Introducing such techniques into the BVH hardware would no doubt further improve the benefits demonstrated in Chapter 4.

Finally, it would be interesting to apply the BVH construction hardware to other applications, such as collision detection, and to the construction of hierarchies for point sets, such as photon maps. In fact, it is easy to envisage applications where the construction of multiple data structures for different components could all be managed by the BVH hardware.

Appendix A

Additional Circuits

This chapter details some additional circuits developed over the course of this research, which are directly applicable to ray-tracing hardware implementations.

A.1 The Ray-Tracing Pipeline

The ray-tracing algorithm is commonly divided into a number of distinct phases, with each one feeding results to the next. The first phase of the pipeline, *ray generation* is responsible for determining the direction and origin of each ray to be cast. The process of generating such rays depends on the algorithm. For example, in Whitted style ray-tracing [Whi80], primary rays are formed by various vector operations related to the camera. In the same algorithm, reflection and refraction rays are generated from intersection points according to the laws of reflection and refraction. In algorithms such as path tracing [Kaj86], the ray directions may be generated randomly. Therefore, this operation is algorithm specific.

The ray generation phase feeds into the second phase, *AABB intersect.* This phase is used as a quick pre-test to filter out any rays which have no potential of intersecting any scene geometry. With some data structures, such as the BVH this phase is built into traversal. However, with kd-trees, which do not explicitly store bounding boxes, it must be performed before traversal.

Those rays that have yielded a valid intersection with the scene AABB are then forwarded to the *traversal* stage. The purpose of the traversal stage is to search the acceleration data structure using the current ray in order to find leaf nodes of the structure which may contain triangles with which the ray intersects. The list of triangles in a leaf node is then forwarded onto the *intersection* stage which find exact intersections (if any) with the primitives in the leaf node. If no intersections are found in a given leaf, the traversal of the ray continues, potentially finding more candidate leaves and attempting further intersections.

If any valid intersections are found, they are forwarded to the shading stage which determines the contribution of a given ray intersection to the relevant pixel colour.

At the early stages of this research, the possibility of investigating hardware traversal and intersection (as opposed to acceleration structure construction) was considered. From this initial work, a number of circuits for some of the stages of the ray-tracing pipeline were created. In particular, circuits for AABB intersection and triangle intersection were created by the author. They are presented here as they may be useful to others wishing to pursue research in hardware ray-tracing.

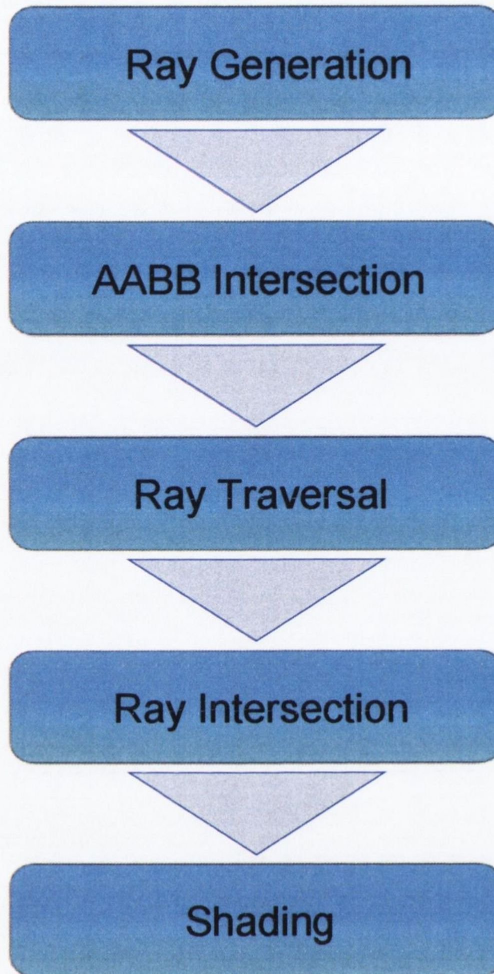


Figure A.1: *The Ray-Tracing Pipeline.*

A.1.1 AABB Intersection Unit

The first question which must be answered before designing any hardware unit is to decide which algorithm it will implement. A number of AABB intersection algorithms for ray-tracing applications are known. Among the most efficient algorithms is that presented by [WBMS05], building on the work of [Smi98].

The hardware implementation of the AABB intersect is shown in Figure A.2. The unit takes in a ray and a bounding box at the top of the pipeline and produces an entry and exit distance, aswell as a valid intersection flag to indicate if an intersection actually occurred. For each axis, a divider calculates the reciprocal of the ray direction in that axis. Concurrently to this operation, the uni-dimensional distance from the origin of the ray to the near and far extents of the AABB is calculated using two subtracters. A comparator is used to determine which of these subtracters represents the near and far distance to the bounding box. The contents of the subtracters are then fed to two multipliers, the other input of which being the reciprocal of the ray. The outputs of the multipliers then represent the $tmin$ and $tmax$ intersection values for the axis.

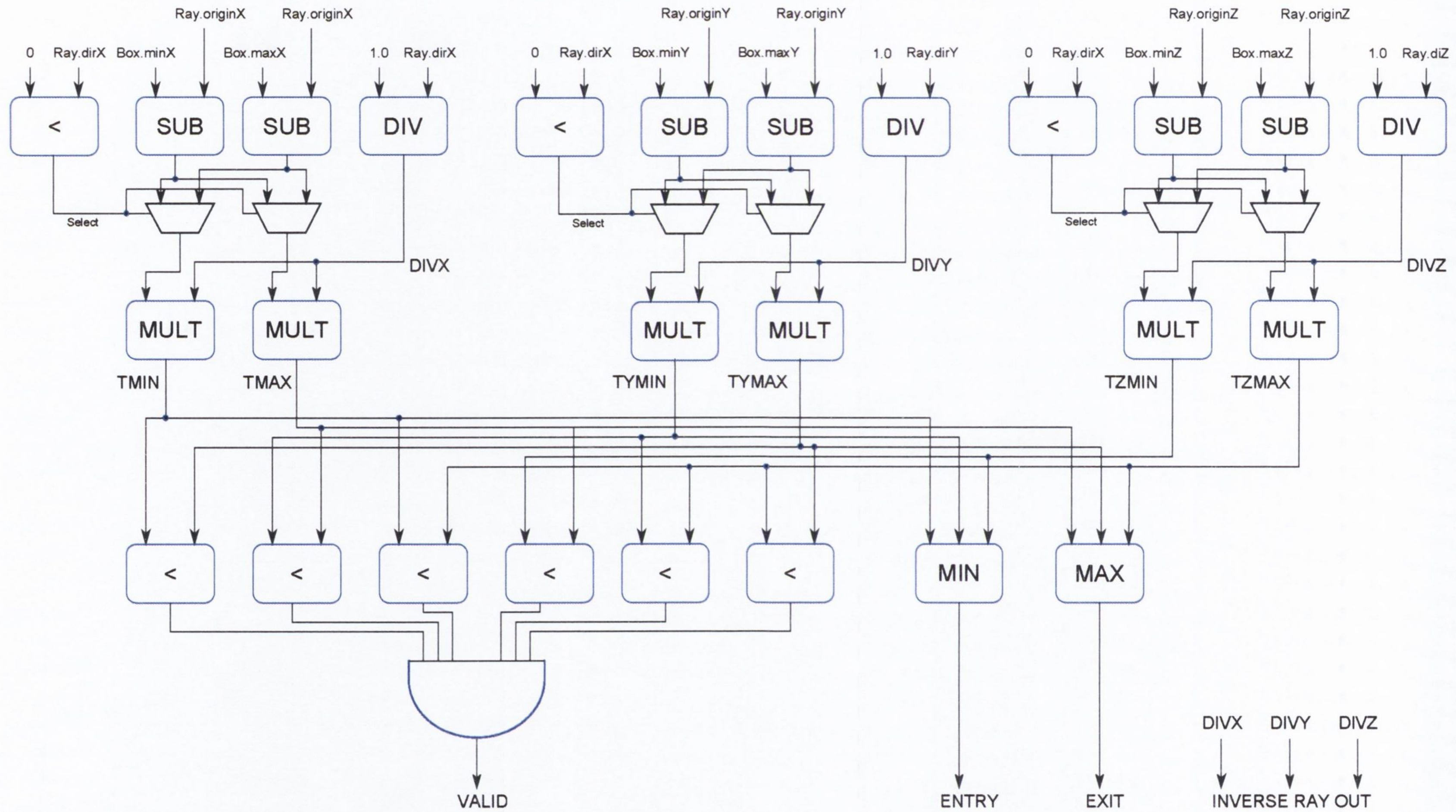


Figure A.2: *Microarchitecture of the AABB intersection Unit.*

A.1.2 Triangle Intersection Unit

As most interactive ray-tracers utilise triangles for scene primitives, a number of ray/triangle intersection algorithms have arisen over the years. In particular, the Möller-Trumbore triangle intersection has been one of the most widely used [MT97]. Figures A.3 and A.4 illustrated a hardware implementation of this algorithm. The intersection unit is fully pipelined and can deliver one ray/triangle intersection per cycle.

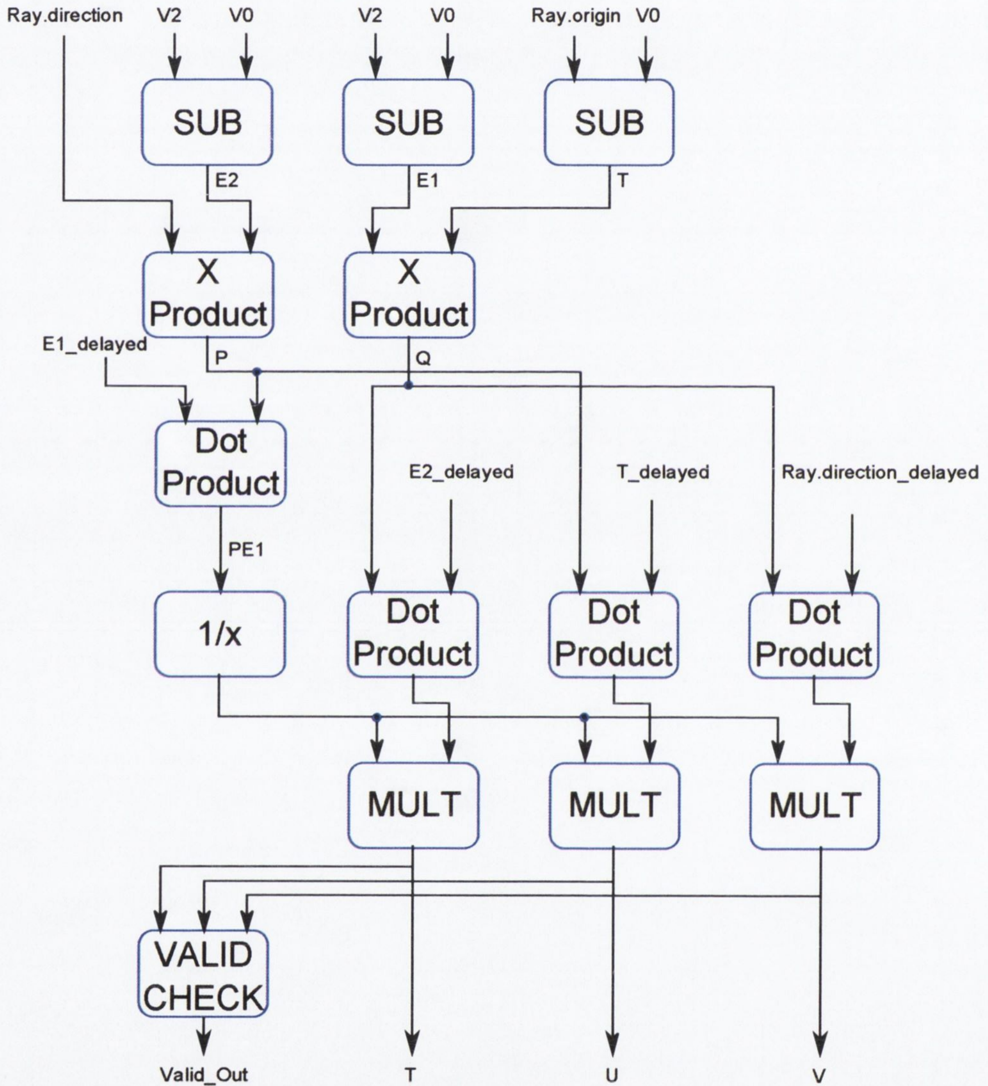


Figure A.3: *Möller-Trumbore Ray Triangle Intersection Unit.*

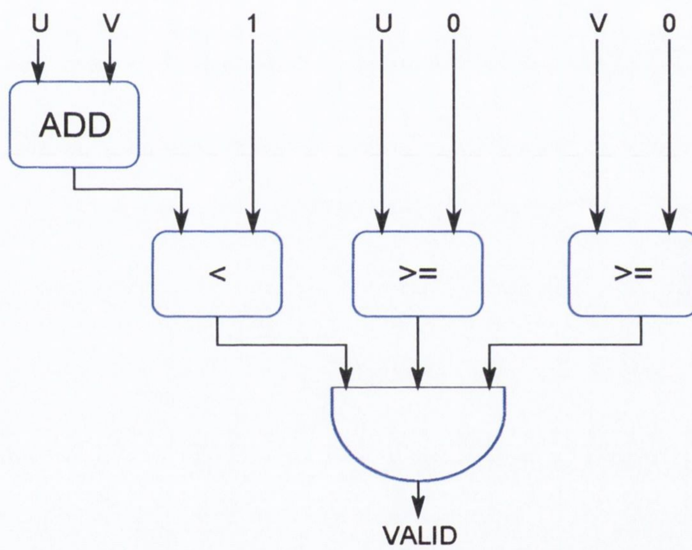


Figure A.4: *Valid Check for Möller-Trumbore Intersection Unit.*

Bibliography

[ACG*92] ABNOUS A., CHRISTENSEN C., GRAY J., LENELL J., NAYLOR A., BAGHERZADEH N.: Design and implementation of the tiny risc microprocessor. *Microprocessors and Microsystems 16*, 4 (1992), 187 – 193.

[AK87] ARVO J., KIRK D.: Fast ray tracing by ray classification. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), SIGGRAPH '87, ACM, pp. 55–64.

[AK90] ARVO J., KIRK D.: Particle transport and image synthesis. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1990), SIGGRAPH '90, ACM, pp. 63–66.

[AK10] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 113–122.

[AKL13] AILA T., KARRAS T., LAINE S.: On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 101–107.

[App68] APPEL A.: Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Comptr. Conf* (1968).

[ATD*02] ANIDO M., TABRIZI N., DU H., SANCHEZ-ELEZ M. M., BAGHERZADEH N.: Interactive ray tracing using a simd reconfigurable architecture. In *Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing* (Washington, DC, USA, 2002), SBAC-PAD '02, IEEE Computer Society, pp. 20–28.

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM 18* (September 1975), 509–517.

[BHWB07] BEYER J., HADWIGER M., WOLFSBERGER S., BUHLER K.: High-quality multimodal volume rendering for preoperative planning of neurosurgical interventions. *Visualization and Computer Graphics, IEEE Transactions on 13*, 6 (nov.-dec. 2007), 1696 –1703.

[Bul13] BULLET: Game Physics Simulation: Home of the open source Bullet Physics Library and physics discussion forums. http://bulletphysics.org/wordpress/, 2013. [Online; accessed 3-January-2013].

[Cad14] Cadence Design Systems: Cadence Design Systems Company Website. http://www.cadence.com/, 2014. [Online; accessed 30-March-2013].

[Cla76] Clark J. H.: Hierarchical geometric models for visible surface algorithms. *Commun. ACM 19*, 10 (1976), 547–554.

[CMHM10] Chung E. S., Milder P. A., Hoe J. C., Mai K.: Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus. In *In MICRO-43: Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture* (2010).

[Coo86] Cook R. L.: Stochastic sampling in computer graphics. *ACM Trans. Graph. 5*, 1 (1986), 51–72.

[CPC84] Cook R. L., Porter T., Carpenter L.: Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 137–145.

[CRR04] Cassagnabère C., Rousselle F., Renaud C.: Path tracing using the ar350 processor. In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (New York, NY, USA, 2004), GRAPHITE '04, ACM, pp. 23–29.

[DCH88] Drebin R. A., Carpenter L., Hanrahan P.: Volume rendering. *SIGGRAPH Comput. Graph. 22*, 4 (June 1988), 65–74.

[DFM12] Doyle M. J., Fowler C., Manzke M.: Hardware Accelerated Construction of SAH-based Bounding Volume Hierarchies for Interactive Ray Tracing. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 209–209.

[DGR*74] Dennard R., Gaensslen F., Rideout V., Bassous E., LeBlanc A.: Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of 9*, 5 (1974), 256–268.

[DMS11] Davidovic T., Marsalek L., Slusallek P.: Performance considerations when using a dedicated ray traversal engine. In *19th International Conference on Computer Graphics, Visualization and Computer Vision 2011 (WSCG 2011) Pilsen* (2 2011), pp. 65–72.

[EBSA*11] Esmaeilzadeh H., Blem E., St. Amant R., Sankaralingam K., Burger D.: Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 365–376.

[EG07] Ernst M., Greiner G.: Early split clipping for bounding volume hierarchies. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on* (2007), pp. 73–78.

[EG08] Ernst M., Greiner G.: Multi bounding volume hierarchies. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug. 2008), pp. 35–40.

[Ern12] Ernst M.: Embree: Photo-realistic ray tracing kernels. http://software.intel.com/en-us/articles/embree-photo-realistic-ray-tracing-kernels, 2012. [Online; accessed 29-March-2013].

[FD09] FABIANOWSKI B., DINGLIANA J.: Interactive global photon mapping. *Computer Graphics Forum 28*, 4 (2009), 1151–1159.

[FR03] FENDER J., ROSE J.: A high-speed ray tracing engine built on a field-programmable system. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on* (Dec. 2003), pp. 188–195.

[FTI86] FUJIMOTO A., TANAKA T., IWATA K.: Arts: Accelerated ray-tracing system. *Computer Graphics and Applications, IEEE 6*, 4 (April 1986), 16–26.

[Gar09] GARANZHA K.: The use of precomputed triangle clusters for accelerated ray tracing in dynamic scenes. *Computer Graphics Forum 28*, 4 (2009), 1199–1206.

[GC91] GORDON D., CHEN S.: Front-to-back display of bsp trees. *IEEE Comput. Graph. Appl. 11*, 5 (Sept. 1991), 79–85.

[GDS*08] GOVINDARAJU V., DJEU P., SANKARALINGAM K., VERNON M., MARK W. R.: Toward a Multicore Architecture for Real-time Ray-tracing. In *Proceedings of the 41st Annual International Symposium on Microarchitecture* (November 2008).

[GHFB13] GU Y., HE Y., FATAHALIAN K., BLELLOCH G.: Efficient bvh construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 81–88.

[GL10] GARANZHA K., LOOP C.: Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum 29*, 2 (2010), 289–298.

[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: Obbtree: a hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 171–180.

[GPBG11] GARANZHA K., PREMOŽE S., BELY A., GALAKTIONOV V.: Grid-based sah bvh construction on a gpu. *The Visual Computer 27* (2011), 697–706.

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 59–64.

[GPR*94] GÜNTHER T., POLIWODA C., REINHART C., HESSER J., MÄNNER R., MEINZER H., BAUR H.: Virim: A massively parallel processor for real-time volume visualization in medicine. In *In Proceedings of the EUROGRAPHICS Workshop on Graphics Hardware '94* (1994).

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007* (Sept. 2007), pp. 113–118.

[GR08] GRIBBLE C., RAMANI K.: Coherent ray tracing via stream filtering. In *In Proceedings of the 2008 IEEE/Eurographics Symposium on Interactive Ray Tracing* (August 2008), pp. 59–66.

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7*, 5 (1987), 14–20.

[GTGB84] Goral C. M., Torrance K. E., Greenberg D. P., Battaile B.: Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), SIGGRAPH '84, ACM, pp. 213–222.

[Hal01] Hall D.: The ar350: Today's ray trace rendering processor. In *In Proceedings of the EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware - Hot 3D Session* (2001).

[Hav00] Havran V.: *Heuristic Ray Shooting Algorithms.* PhD thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, 2000.

[HFFA11] Hardavellas N., Ferdman M., Falsafi B., Ailamaki A.: Toward dark silicon in servers. *Micro, IEEE 31*, 4 (2011), 6–15.

[HK07] Hanika J., Keller A.: Towards hardware ray tracing using fixed point arithmetic. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on* (2007), pp. 119–128.

[HKB80] H.Fuchs, Kedem Z., B.Naylor: On visible surface generation by a priori tree structures. In *Proc. SIGGRAPH '80* (1980).

[Hum96] Humphreys G.: *Tigershark: A hardware accelerated ray-tracing engine.* Tech. rep., Princeton University, 1996.

[IEE08] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008* (2008), 1–70.

[Ima10] Imagination Technologies: Imagination technologies plc acquisition announcement. http://www.imgtec.com/corporate/newsdetail.asp?NewsID=602, 2010. [Online; accessed 15-November-2012].

[Ima12] Imagination Technologies: Caustic by Imagination Product Website. https://caustic.com/, 2012. [Online; accessed 15-November-2012].

[Jac13] Jacco Bikker: Brigade: Real-time Path Tracing. http://igad.nhtv.nl/~bikker/, 2013. [Online; accessed 30-October-2013].

[Jen96] Jensen H. W.: Global illumination using photon maps. In *In Eurographics Workshop on Rendering* (1996), pp. 21–30.

[JLBM05] Johnson G. S., Lee J., Burns C. A., Mark W. R.: The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph. 24*, 4 (Oct. 2005), 1462–1482.

[KA13] Karras T., Aila T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 89–99.

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

[Kar12] Karras T.: Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *High Performance Graphics* (2012), pp. 33–37.

[KBS11] Kalojanov J., Billeter M., Slusallek P.: Two-level grids for ray tracing on gpus. *Computer Graphics Forum 30*, 2 (2011), 307–314.

[Ken08] Kensler A.: Tree rotations for improving bounding volume hierarchies. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (2008), pp. 73–76.

[KHM*98] Klosowski J. T., Held M., Mitchell J. S. B., Sowizral H., Zikan K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics 4* (1998), 21–36.

[KIS*12] Kopta D., Ize T., Spjut J., Brunvand E., Davis A., Kensler A.: Fast, effective bvh updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 197–204.

[KK86] Kay T. L., Kajiya J. T.: Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), ACM, pp. 269–278.

[KKK10] Kim H.-Y., Kim Y.-J., Kim L.-S.: Reconfigurable mobile stream processor for ray tracing. In *Custom Integrated Circuits Conference (CICC), 2010 IEEE* (sept. 2010), pp. 1–4.

[KKK12] Kim H.-Y., Kim Y.-J., Kim L.-S.: Mrtp: Mobile ray tracing processor with reconfigurable stream multi-processors for high datapath utilization. *Solid-State Circuits, IEEE Journal of 47*, 2 (feb. 2012), 518 –535.

[KKW*13] Keller A., Karras T., Wald I., Aila T., Laine S., Bikker J., Gribble C., Lee W.-J., McCombe J.: Ray tracing is the future and ever will be... In *ACM SIGGRAPH 2013 Courses* (New York, NY, USA, 2013), SIGGRAPH '13, ACM, pp. 9:1–9:7.

[KŁLH12] Kim J.-W., Lee W.-J., Lee M.-W., Han T.-D.: Parallel-pipeline-based traversal unit for hardware-accelerated ray tracing. In *SIGGRAPH Asia 2012 Posters* (New York, NY, USA, 2012), SA '12, ACM, pp. 42:1–42:1.

[KS97] Knittel G., Strasser W.: Vizard: Visualization accelerator for realtime display. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 1997), HWWS '97, ACM, pp. 139–146.

[KSBD10] Kopta D., Spjut J. B., Brunvand E., Davis A.: Efficient mimd architectures for high-performance ray tracing. In *ICCD* (2010), IEEE, pp. 9–16.

[KSS*01] Kobayashi H., Suzuki K., Sano K., Kaeriyama Y., Saida Y., Oba N., Nakamura T.: 3dcgiram: an intelligent memory architecture for photo-realistic image synthesis. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on* (2001), pp. 462–467.

[KSS*13] Kopta D., Shkurko K., Spjut J., Brunvand E., Davis A.: An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 121–128.

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D.: Fast bvh construction on gpus. *Comput. Graph. Forum 28*, 2 (2009), 375–384.

[LK11] LAINE S., KARRAS T.: High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 79–88.

[LLN*12] LEE W.-J., LEE S.-H., NAH J.-H., KIM J.-W., SHIN Y., LEE J., JUNG S.-Y.: Sgrt: a scalable mobile gpu architecture based on ray tracing. In *ACM SIGGRAPH 2012 Posters* (New York, NY, USA, 2012), SIGGRAPH '12, ACM, pp. 44:1–44:1.

[LSL*13] LEE W.-J., SHIN Y., LEE J., KIM J.-W., NAH J.-H., JUNG S., LEE S., PARK H.-S., HAN T.-D.: Sgrt: a mobile gpu architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 109–119.

[Lux12] LUXRENDER: LuxRender Software Wesbite. http://www.luxrender.net/, 2012. [Online; accessed 18-November-2012].

[LW93] LAFORTUNE E. P., WILLEMS Y. D.: Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques* (1993), Compugraphics '93, pp. 145–153.

[LYTM06] LAUTERBACH C., YOON S.-E., TUFT D., MANOCHA D.: Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *Interactive Ray Tracing 2006, IEEE Symposium on* (sept. 2006), pp. 39 –46.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6* (1990), 153–166. 10.1007/BF01911006.

[Men12] MENTOR GRAPHICS: Mentor Graphics Company Website. http://www.mentor.com/, 2012. [Online; accessed 10-December-2013].

[MKS98] MEISSNER M., KANUS U., STRASSNER W.: Vizard ii, a pci-card for real-time volume rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware* (New York, NY, USA, 1998), HWWS '98, ACM, pp. 61–67.

[MT97] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray-triangle intersection. *J. Graph. Tools 2*, 1 (1997), 21–28.

[NPK*10] NAH J.-H., PARK J.-S., KIM J.-W., PARK C., HAN T.-D.: Ordered depth-first layouts for ray tracing. In *ACM SIGGRAPH ASIA 2010 Sketches* (New York, NY, USA, 2010), SA '10, ACM, pp. 55:1–55:2.

[NPP*11] NAH J.-H., PARK J.-S., PARK C., KIM J.-W., JUNG Y.-H., PARK W.-C., HAN T.-D.: T&i engine: traversal and intersection engine for hardware accelerated ray tracing. In *Proceedings of the 2011 SIGGRAPH Asia Conference* (New York, NY, USA, 2011), SA '11, ACM, pp. 160:1–160:10.

[NVI11] NVIDIA: Bringing high-end graphics to handheld devices. *NVIDIA White Paper* (2011).

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W. R.: Large ray packets for real-time whitted ray tracing. In *IEEE/EG Symposium on Interactive Ray Tracing (IRT)* (Aug 2008), pp. 41—-48.

[OS07] Ochsenfahrt U., Salomon R.: Crema: A parallel hardware raytracing machine. In *ISCAS* (2007), IEEE, pp. 769–772.

[PBD*10] Parker S. G., Bigler J., Dietrich A., Friedrich H., Hoberock J., Luebke D., McAllister D., McGuire M., Morley K., Robison A., Stich M.: Optix: a general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July 2010), 66:1–66:13.

[PHK*99] Pfister H., Hardenbergh J., Knittel J., Lauer H., Seiler L.: The volumepro real-time ray-casting system. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 251–260.

[PhNP*08] Park W.-C., ho Nah J., Park J.-S., Lee K.-H., Kim D.-S., Kim S.-D., Park J.-H., Kim C.-G., Kang Y.-S., Yang S.-B., Han T.-D.: An fpga implementation of whitted-style ray tracing accelerator. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug. 2008), pp. 187–187.

[PL10] Pantaleoni J., Luebke D.: Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 87–95.

[PMSP12] Peterson L., McCombe J., Salsbury R., Purcell S.: Architectures for parallelized intersection testing and shading for ray-tracing rendering. US Patent Number 8203559, 2012.

[PSGC10] Peterson L., Salsbury R., Gies S., Clohset S.: Dynamic ray population control. US Patent Application Number 12/771,408, 2010.

[RDK*00] Rixner S., Dally W., Kapasi U., Mattson P., Owens J.: Memory access scheduling. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on* (2000), pp. 128–138.

[Res07] Reshetov A.: Faster ray packets - triangle intersection through vertex culling. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on* (Sept 2007), pp. 105–112.

[RGD09] Ramani K., Gribble C. P., Davis A.: Streamray: a stream filtering architecture for coherent ray tracing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS '09, ACM, pp. 325–336.

[RHAZ06] Raabe A., Hochgurtel S., Anlauf J., Zachmann G.: Space-efficient fpga-accelerated collision detection for virtual prototyping. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings* (march 2006), vol. 2, p. 6 pp.

[RSH05] Reshetov A., Soupikov A., Hurley J.: Multi-level ray tracing algorithm. In *ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), SIGGRAPH '05, ACM, pp. 1176–1185.

[RW80] Rubin S. M., Whitted T.: A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph. 14*, 3 (1980), 110–116.

[SBK*08] Spjut J. B., Boulos S., Kopta D., Brunvand E., Kellis S.: TRaX: A Multi-Threaded Architecture for Real-Time Ray Tracing. In *Symposium on Application Specific Processors* (2008), pp. 108–114.

[SBU11] Sopin D., Bogolepov D., Ulyanov D.: Real-time sah bvh construction for ray tracing dynamic scenes. In *Proceedings of Graphicon 2011* (Moscow, Russia, September 2011).

[SDL13] SDL: Simple DirectMedia Layer (SDL). http://www.libsdl.org/, 2013. [Online; accessed 10-December-2013].

[SEDT*03] Sanchez-Elez M., Du H., Tabrizi N., Long Y., Bagherzadeh N., Fernández M.: Algorithm optimizations and mapping scheme for interactive ray tracing on a reconfigurable architecture. *Computers & Graphics 27*, 5 (2003), 701–713.

[SG93] Schreiner S., Galloway R.L. J.: A fast maximum-intensity projection algorithm for generating magnetic resonance angiograms. *Medical Imaging, IEEE Transactions on 12*, 1 (mar 1993), 50 –57.

[SKBD12] Spjut J., Kopta D., Brunvand E., Davis A.: A mobile accelerator architecture for ray tracing. In *In the 3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW)* (Feb. 2012).

[SKKB09] Spjut J., Kensler A., Kopta D., Brunvand E.: Trax: a multicore hardware architecture for real-time ray tracing. *Trans. Comp.-Aided Des. Integ. Cir. Sys. 28*, 12 (Dec. 2009), 1802–1815.

[SLS03] Schmittler J., Leidinger A., Slusallek P.: A virtual memory architecture for real-time ray tracing hardware. *Computers and Graphics 27*, 5 (2003), 693 – 699.

[Smi98] Smits B.: Efficiency issues for ray tracing. *Journal of Graphics Tools: JGT*, 3 (1998).

[SPLB97] Santarelli M., Positano V., Landini L., Benassi A.: Volume rendering in medicine: the role of image coherence. In *Computers in Cardiology 1997* (sep 1997), pp. 323 –326.

[SSK07] Shevtsov M., Soupikov A., Kapustin A.: Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum 26*, 3 (2007), 395–404.

[Sta13] Stanford Computer Graphics Laboratory: The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/, 2013. [Online; accessed 12-November-2013].

[SWS02] Schmittler J., Wald I., Slusallek P.: Saarcor: A hardware architecture for ray tracing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 27–36.

[SWW*04] Schmittler J., Woop S., Wagner D., Paul W. J., Slusallek P.: Realtime ray tracing of dynamic scenes on an fpga chip. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM, pp. 95–106.

[SZ09] Steffen M., Zambreno J.: Design and evaluation of a hardware accelerated ray tracing data structure. In *In Proceedings of EG UK Theory and Practice of Computer Graphics* (2009), pp. 101–108.

[SZ10] Steffen M., Zambreno J.: A hardware pipeline for accelerating ray traversal algorithms on streaming processors. In *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on* (june 2010), pp. 22 –29.

[Tay12] Taylor M. B.: Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 1131–1136.

[TL01] Todman T., Luk W.: Reconfigurable designs for ray tracing. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on* (2001), pp. 300–301.

[TMCS08] Toksvig M., Mathieson J., Cabral B., Smith B.: Nvidia tegra: Enabling stunning handheld graphics and hd video. In *In Proceedings of Hot Chips 20 (HC20)* (2008).

[Uni13] University of Utah, SCI Institute: The Utah Animation Repository. http://www.sci.utah.edu/~wald/animrep/, 2013. [Online; accessed 12-November-2013].

[vdB98] van den Bergen G.: Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools 2*, 4 (Jan. 1998), 1–13.

[VG94] Veach E., Guibas L.: Bidirectional estimators for light transport. In *Proceedings of the Fifth Eurographics Workshop on Rendering* (June 1994), Eurographics, pp. 147–162.

[VG95] Veach E., Guibas L. J.: Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1995), SIGGRAPH '95, ACM, pp. 419–428.

[VG97] Veach E., Guibas L. J.: Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 65–76.

[VSG*10] Venkatesh G., Sampson J., Goulding N., Garcia S., Bryksin V., Lugo-Martinez J., Swanson S., Taylor M. B.: Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (New York, NY, USA, 2010), ASPLOS '10, ACM, pp. 205–218.

[Wal07] Wald I.: On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (2007).

[Wal12] Wald I.: Fast construction of sah bvhs on the intel many integrated core (mic) architecture. *Visualization and Computer Graphics, IEEE Transactions on 18*, 1 (2012), 47–57.

[WBB08] Wald I., Benthin C., Boulos S.: Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs -. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (2008), pp. 49–57.

[WBKP08] Walter B., Bala K., Kulkarni M., Pingali K.: Fast agglomerative clustering for rendering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug. 2008), pp. 81–86.

[WBMS05] Williams A., Barrus S., Morley R. K., Shirley P.: An efficient and robust ray-box intersection algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM, p. 9.

[WBS03] Wald I., Benthin C., Slusallek P.: Distributed interactive ray tracing of dynamic scenes. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Washington, DC, USA, 2003), IEEE Computer Society, p. 11.

[WBS06] Woop S., Brunvand E., Slusallek P.: Estimating performance of a ray-tracing asic design. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (September 2006), pp. 7–14.

[WBS07] Wald I., Boulos S., Shirley P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (Jan. 2007).

[WH06] Wald I., Havran V.: On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 61–70.

[WHG84] Weghorst H., Hooper G., Greenberg D. P.: Improved computational methods for ray tracing. *ACM Trans. Graph. 3*, 1 (1984), 52–69.

[Whi80] Whitted T.: An improved illumination model for shaded display. *Commun. ACM 23*, 6 (1980), 343–349.

[WIK*06] Wald I., Ize T., Kensler A., Knoll A., Parker S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph. 25*, 3 (2006), 485–493.

[WK06] Wächter C., Keller A.: Instant ray tracing: the bounding interval hierarchy. In *Proceedings of the 17th Eurographics conference on Rendering Techniques* (Aire-la-Ville, Switzerland, Switzerland, 2006), EGSR'06, Eurographics Association, pp. 139–149.

[WMS06] Woop S., Marmitt G., Slusallek P.: B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (New York, NY, USA, 2006), ACM, pp. 67–77.

[Woo06] Woop S.: *DRPU: A Programmable Hardware Architecture for Real-time Ray Tracing of Coherent Dynamic Scenes.* PhD thesis, Saarland University, 2006.

[WRC88] Ward G. J., Rubinstein F. M., Clear R. D.: A ray tracing solution for diffuse interreflection. *SIGGRAPH Comput. Graph. 22* (June 1988), 85–92.

[WSBW01] Wald I., Slusallek P., Benthin C., Wagner M.: Interactive rendering with coherent ray tracing. In *Computer Graphics Forum* (2001), pp. 153–164.

[WSS05] Woop S., Schmittler J., Slusallek P.: Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph. 24*, 3 (July 2005), 434–444.

[YCM07] Yoon S.-E., Curtis S., Manocha D.: Ray tracing dynamic scenes using selective restructuring. In *ACM SIGGRAPH 2007 sketches* (New York, NY, USA, 2007), SIGGRAPH '07, ACM.

[ZHWG08] Zhou K., Hou Q., Wang R., Guo B.: Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27* (December 2008), 126:1–126:11.

[ZK03] Zachmann G., Knittel G.: *High-Performance Collision Detection Hardware.* Tech. Rep. CG-2003-3, University Bonn, Informatikk II, Bonn, Germany, Aug. 2003.

©

Michael J. Doyle

2014