University of Dublin, Trinity College

Approximate Agglomerative Clustering in the Intel Embree Ray Tracing Kernels

Author:
Alan Cross

Supervisor:
Dr. Michael Manzke

A dissertation submitted in fulfilment of the requirements for the Degree of Master in Computer Science

Submitted to the University of Dublin, Trinity College

May 2014

Declaration of Authorship

I, Alan Cross, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University, and that the library may lend or copy any part thereof on request.

Signed: _____

Date: _____

Summary

Ray tracing algorithms have long since been known as techniques capable of producing very high degrees of realism in graphics, but at significant computational cost. As a result, there has been a high degree of research spent on the acceleration of these algorithms. One such area of research has been in the construction of Bounding Volume Hierarchies (BVHs). These data structures are used in conjuction with ray tracing and collision detection algorithms to drastically increase performance. They accomplish this by helping to eliminate potential intersection candidates within a scene by omitting geometric objects located in bounding volumes which are not intersected by the current ray.

The fast and efficient construction of BVHs has been the subject of extensive research as a result of their effectiveness. One relatively unstudied construction technique is Agglomerative Clustering and in particular Approximate Agglomerative Clustering (AAC). This research presents an implementation of the algorithm in the Intel Embree Ray Tracing Kernels, which utilise the power of modern multicore CPUs and SPMD techniques. Recent research advocates the exploration and comparison of new construction techniques and this implementation shows AAC to be a viable option which demonstrates many advantages over other solutions.

Acknowledgements

I would like to thank both my supervisor Dr. Michael Manzke for his continued advice and guidance throughout my research and Michael Doyle for his help and recommendations.

I would also like to thank the Embree team at Intel, Ingo Wald, Sven Woop and Carsten Benthin, for their help in understanding the kernels and with any project queries I had.

Finally, I would like to thank my family, friends and girlfriend for their continued support and understanding throughout my time in college.

Alan Cross

Contents

Declaration of Authorship i

Summary ii

Acknowledgements iii

Contents iv

List of Figures vi

List of Tables vii

List of Listings viii

Abbreviations ix

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Motivation . . . 3
- 1.3 Aims and Objectives . . . 3
- 1.4 Dissertation Structure . . . 3

2 State of the Art 4
- 2.1 Acceleration Structures . . . 5
  - 2.1.1 Kd-trees . . . 6
  - 2.1.2 Grids . . . 6
  - 2.1.3 Bounding Volume Hierarchy . . . 7
- 2.2 Construction Techniques . . . 8
  - 2.2.1 Top-down . . . 8
  - 2.2.2 Bottom-up . . . 10
  - 2.2.3 Insertion . . . 10
- 2.3 Approaches . . . 11
  - 2.3.1 Construction on Multicore CPUs . . . 11
  - 2.3.2 Construction on Manycore CPUs . . . 12
  - 2.3.3 Construction on GPUs . . . 12
  - 2.3.4 Construction on Dedicated Acceleration Architectures . . . 13

3 Experimental Design and Implementation 15
3.1 Embree Renderer . . . 15
3.2 Embree Ray Tracing Kernels . . . 17
3.2.1 Primitives and Data Structures . . . 17
3.2.2 Traversal and Intersection Kernels . . . 18
3.2.3 BVH Construction Kernels . . . 20
3.2.4 Embree Kernel API . . . 21
3.3 Approximate Agglomerative Clustering . . . 22
3.4 Implementation . . . 27
3.4.1 Parallelisation . . . 30
3.4.2 Integration with the Kernels . . . 32
3.4.3 Kernel Features Utilisation . . . 33

4 Evaluation 34
4.1 BVH Build Times . . . 37
4.2 BVH Quality . . . 37
4.3 Discussion . . . 38

5 Conclusion 43
5.1 Future Work . . . 43
5.2 Conclusion . . . 44

Bibliography 45

List of Figures

1.1 BVH Structure 2

3.1 Embree Device Interface 16
3.2 Quad BVH (BVH4) 18
3.3 AAC Constraint Tree 24
3.4 Morton Codes 24
3.5 AAC Derivation Tree 31

4.2 Imperial Crown of Austria & Scene Render 36
4.3 Dragon Bust B & Cornell Box Scene Render 36
4.4 BVH Build Times (AAC) 38
4.5 BVH Build Times 39
4.6 BVH Build Quality 40
4.7 BVH Build Quality (Leaves) 41

List of Tables

4.1 Model Complexity 35
4.2 BVH Build Times (AAC) 38
4.3 BVH Build Times 39
4.4 BVH Build Quality 40
4.5 BVH Build Quality (Leaves) 41

List of Listings

3.1 Beginning of Downward Phase 28
3.2 Primitive and Node Splitting 28
3.3 Cluster Initialisation and Beginning of Bottom-up Clustering 29
3.4 Sub-tree Thread Assignment. 32

Abbreviations

BVH	**B**ounding **V**olume **H**ierarchy
AAC	**A**pproximate **A**gglomerative **C**lustering
SPMD	**S**ingle **P**rogram **M**ultiple **D**ata
SAH	**S**urface **A**rea **H**euristic
AABB	**A**xis **A**ligned **B**ounding **B**ox
ISPC	**I**ntel **S**PMD **P**rogram **C**ompiler

Chapter 1

Introduction

1.1 Background

Ray tracing algorithms are known for producing highly realistic images, but at a significant computational cost. Ray tracing generates its images by tracing the light through pixels in an image plane and simulating the effects of its encounters with virtual objects. Very high visual realism, usually higher than scanline and rasterization rendering methods, can be achieved but at this greater computational cost. It is this cost that has led to the research and implementation of various acceleration structures such as Bounding Volume Hierarchies (BVHs).

BVHs are tree structures based on a set of geometric objects in a scene. Each object is wrapped in "bounding volumes" that form the leaf nodes of the tree. These nodes are then grouped as small sets and enclosed within larger bounding volumes. These, in turn, are also grouped and enclosed within other larger bounding volumes in a recursive fashion until a single bounding volume has been reached at the top of the tree (see Figure 1.1). These data structures are then used in conjuction with ray tracing and collision detection algorithms to drastically increase performance. They accomplish this by helping to eliminate potential intersection candidates within a scene by omitting geometric objects located in bounding volumes, which are not intersected by the current ray. The spatial map provided by the structure is used for quickly culling away superfluous intersection tests. Even though the structures must be rebuilt and updated

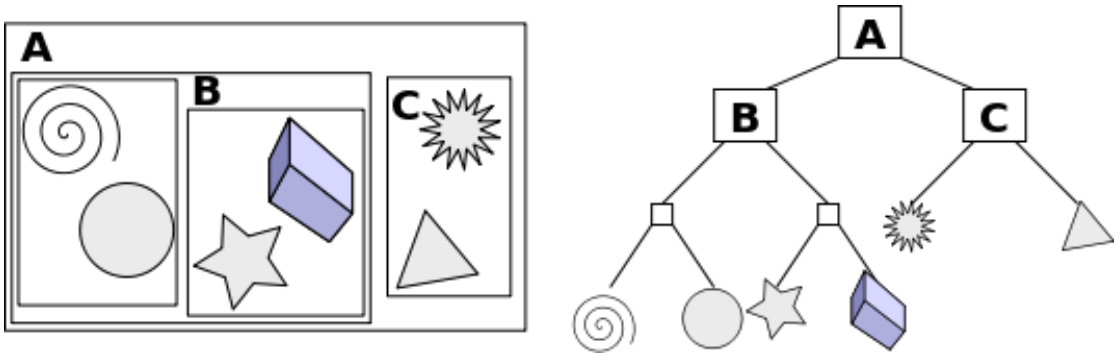


FIGURE 1.1: BVH Structure. The objects at the bottom of the image on the right hand side represent the leaf nodes of the tree, and the smallest bounding volume boxes. 'A' represents the root of the tree with 'B' and 'C' being the disjoint subset of the root node. [34]

over time while rendering dynamic scenes (the spatial map provided by the structure is invalidated by scene motion), the efficiency of the structures has made them essential in any interactive ray tracing system.

As building these BVHs represents a large proportion of the total time to image, there has been a great deal of research directed at the goal of faster and more efficient BVH construction. Much research has been on parallel construction on both multicore and manycore platforms [3, 19, 31], where multicores represent a single component with two or more independent processors or "cores" and manycores where the number of cores is large enough (tens/hundreds/thousands of cores) that traditional multi-processor techniques are no longer efficient. Research has demonstrated that a great deal of parallelism is available and larger performance improvements over serial algorithms are achievable. Other approaches such as specialised hardware devices have also achieved high performance improvements by design of dedicated microarchitectures for the construction of acceleration structures [3].

The construction algorithm bears as much importance as the architecture it is being constructed on. Different algorithms such as the binned SAH (Surface Area Heuristic), Spatial Splits or Morton builds are suited to different architectures and different scenes.

1.2 Motivation

As research continues to pursue real time ray tracing, there is continued emphasis on fast BVH construction. The Intel Embree ray tracing kernels are open-source and provide a framework for fast ray tracing on x86 CPUs and incorporate a number of BVH construction algorithms and approaches. The kernels provide a great starting point to implement new construction algorithms and analyse their efficiency and prospects.

1.3 Aims and Objectives

Approximate Agglomerative Clustering (AAC) is a relatively new and unstudied BVH construction algorithm. This research presents an implemention of the algorithm in the Embree Ray Tracing Kernels, taking advantage of the kernel's highly optimised CPU utilisation techniques. The effectiveness and potential of the algorithm within the kernels will be observed and evaluated as well as how it may utilise the kernel's features in order to contend with other more established construction techniques.

1.4 Dissertation Structure

This report is structured as follows: Chapter 2 provides an outline and discussion of previous BVH construction approaches and algorithms and the current state of research. The third chapter presents the Embree Ray Tracer and Kernels as well as the experimental design and implementation of the AAC algorithm. The fourth chapter provides an evaluation of the implementation and the algorithm. In the final chapter, Chapter 5, future work in the area is discussed and final thoughts and conclusions are offered.

Chapter 2

State of the Art

BVHs have been extensively used for ray tracing and collision detection. This can be attributed to the fact that they have been proven to represent a good compromise between traversal performance and construction time. In addition, fast refitting techniques are available for BVHs [18], making them highly suitable for deformable geometry.

BVHs are typically constructed of binary tress where each node of the tree represents a bounding volume. The bounding volumes usually take the form of an Axis-Aligned Bounding Box (AABB) which bounds some subset of the scene geometry. The AABB corresponding to the root node of the tree bounds the entire scene. The two child nodes of the root node bound disjoint subsets of the scene, and each scene primitive will be present in exactly one of the children. The two child nodes can be recursively subdivided in a similar fashion until a termination criterion is met. Typical termination criterion involves terminating at a certain number of primitives, or at a maximum tree depth.

For ray tracing, many BVH construction algorithms follow a top-down procedure. Starting with the root node which bounds the entire scene, nodes are split according to a splitting scheme and child nodes are created. These nodes are further subdivided until a leaf node is reached. The choice of how to split the nodes can have a profound effect on rendering efficiency.

This section creates a context for the work of this dissertation by providing details of the research to date on ray tracing acceleration structures, how they are constructed and in

what sort of environments or architectures. This context and background research will allow for a meaningful demonstration and evaluation of work in Chapters 3 and 4.

2.1 Acceleration Structures

Researchers have been achieving interactive ray tracing as early as the 1990s by using massively parallel supercomputers [10, 21, 23]. Fortunately, the growing capabilities of modern computers has meant that this compute power is becoming more widely available and accessible through commodity architectures like GPUs and multi-core CPUs. Since Reshetov et al.'s "Multilevel Ray Traversal" [24] in particular, ray tracers have been able to achieve fully interactive frame rates in non-trivial scenes on multi-core desktop PCs. Subsequently, near real-time performance in ray tracing has been demonstrated on various different architectures and with a varety of different data structures and traversal algorithms [29].

For ray tracers to achieve fast, real-time performance, they depend on the use of efficient spatial data structures such as kd-trees, grids and BVHs. Although there has been a long running debate as to which data structure is the best, by 2005, most fast ray tracers were using the kd-tree. As kd-trees are quite costly to build and are not easily incrementally updated, they are not very well suited to handling dynamic or animated scenes. As a result, a lot of research has been spent on producing improved and more efficient data structures which may support dynamic scenes.

This research has led to improved ray tracing of dynamic scenes by using faster and more efficient data strcutures like the kd-tree, grids and BVHs. These different structures share various performance-vs-flexibility tradeoffs. Of the three dominant ray tracing data structures, grids are currently considered to be the fastest structure to build but are somewhat less efficient for traversal and intersection. Kd-trees are considered the most efficient for traversal and intersection but are the most costly in terms of construction. BVHs lie somewhere in between in terms of build time and are close to kd-trees for traversal performance and so are gaining prominance and favour in ray tracing solutions and research.

2.1.1 Kd-trees

Kd-trees are a space partitioning data structure which organises points in a k-dimensional space. The structure takes the form of a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts known as half-spaces. Points to the left of the hyperplane are represented by the left subtree of that node while points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen by first associating each node in the tree with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. Then, for example, if for a particular split the "x" axis is chosen, points in the subtree with a smaller "x" value than the node's will appear in the left subtree and all points with a larger "x" value will be in the right subtree. The hyperplane would also be set by the x-value of the point, and its normal would be the unit x-axis. Kd-trees have been well studied for a long time and a lot of optimised approaches to replace the simple median splitting construction method have been developed. The most popular of which involves partitioning of nodes by the SAH [9]. Further research has been undertaken to optimize the SAH kd-tree construction in areas related to performing sorted-order coordinate sweeps to compute the SAH and maintaining this sorted order as the bounding boxes are moved and subdivided [28].

2.1.2 Grids

Another spatial acceleration data structure are grids [1] which partition the space itself into regions or voxels determined by a grid spacing metric. Each voxel has a list of objects from the scene that are in the voxel. If an object spans several voxels the object is in more than one list. When a ray is shot, the voxel where it originates is first inspected. If the ray hits any objects inside the starting voxel list, the intersections are sorted and the closest one is retained. If the intersection is in the current voxel there is no need to intersect any other objects as the closest intersection has been found. If no intersection is found in the current voxel or the object list is empty, the ray tracing is continued and the ray is followed into a neighbouring voxel and that voxel's object list is checked. Ray tracing continues in this fashion until either an intersection is found, or the space

partition is completely traversed. Since objects are intersected in roughly the order as they occur along the ray and trivially reject objects which are far away from the ray, the number of intersections that need to be performed is vastly reduced. Research has been conducted on improving the representation and building of grid stuctures in recent years. Compact and hashed representations and builds for the structures have been shown to take several important advantages over alternative acceleration structures such as "short build time, a short time to image, robust ray traversal, and easy implementation" [17].

2.1.3 Bounding Volume Hierarchy

BVHs are tree structures over a set of geometric objects. The leaf nodes encompass single geometric objects in "bounding volumes". The leaf nodes are grouped together as small sets and are enclosed within larger bounding volumes. This grouping continues until one node is left, representing the root of the tree and a single bounding volume (see Figure 1.1). Although BVHs are efficient, they can be quite dificult to build. This has led to efforts to avoid full rebuilding of the structure by relying on refitting the BVH [18, 30]. Other proposed solutions include coupling refitting techniques with selective restructuring [36] or infrequent/asynchronous rebuilding [13, 18]. An area lacking in research is fast rebuilding of BVHs from scratch. Solutions such as Wächter et al.'s [27] fast spatial-median build have been proposed. The build was originally intended for s-kd-tree (spatial kd-tree) like Bounding Interval Hierarchies (BIH) but it also applies for BVHs. Despite allowing for fast rebuilds, the algorithm gives somewhat reduced traversal performance as an SAH is not used to determine how to best build the hierarchy, but instead splits are determined at the spatial median. Several properties are desirable in an efficient BVH when designing for an application [4]:

- The nodes contained in any given subtree should be near each other. The lower down the tree the nodes are, the nearer they should be together.
- Each node in the hierarchy should be of minimal volume.
- The sum of all bounding volumes should be minimal.

- Greater attention should be paid to nodes near the root of the hierarchy. Pruning a node near the root of the tree removes objects from further consideration than removal of a deeper node would.
- The volume of overlap of sibling nodes should be minimal.
- The hierarchy should be balanced with respect to both its node structure and its content. Balancing allows as much of the hierarchy as possible to be pruned whenever a branch is not traversed into.

A BVH bearing these desirable properties may demonstrate more efficient build and traversal times.

Other ray tracing acceleration structures may include BIHs (Bounding Interval Hierarchies) or BSP (Binary Space Partitioning) trees but the BVH will be the structure of choice and focus for this dissertation.

2.2 Construction Techniques

There are three main categories of tree construction methods: top-down, bottom-up and insertion methods. Both top-down and bottom-up techniques are considered to be "off-line" methods as they require all scene primitives to be available before construction starts. Some construction techniques can also be defined as either high-quality builders or as fast builders, with each opting either for structure quality or the speed of the build, although current research seeks to parallelise and speed up high quality build techniques to get the best of both [14, 31].

2.2.1 Top-down

Top-down methods of tree construction begin by partitioning the input set into two or more subsets. These subsets are then bounded. Subsets are recursively partititioned, and bounded, into further subsets until subsets contain single primitives i.e. the leaf nodes are reached. The resulting tree root contains the bounds for the full scene, with subsets bounding smaller and smaller areas of the scene. Top-down methods are easy

to implement, fast to construct and the most popular choice of construction techniques but they do not result in the best possible trees in general.

Top-down build procedures are one of the most researched areas in BVH construction with one of the most well known methods being the top-down, greedy surface area heuristic build [20] which tries to maximise traversal efficiency by minimising the SAH cost at every step. The most widely adopted simplification of the full top-down partitioning build is the binned SAH approach [29]. The build procedure begins with an input of N primitives in a sub-tree that covers a 3D volume V. Assuming the subtree gets partitioned into two halves L and R with number of triangles N_L and N_R and with associated volumes V_L and V_R respectively, the traversal cost can be estimated as:

$$Cost(V \rightarrow L, R) = K_T + K_I(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R) \tag{2.1}$$

where $SA(V)$ represents the surface area of V, and K_T and K_I are some constants for the estimated traversal step cost and triangle intersection [29].

The cost estimate is used to perform a greedy SAH build where possible partitions are evaluated and the one with the lowest expected cost is selected and then recursed. For a BVH, the "split" partitions the set of triangles into two subsets. As there are $2^N - 2$ possible partitions, testing them all would be intractable and so the build looks at parititioning the triangles using planes similiar to the ones used for kd-trees to reduce the number of partition tests. The planes make it possible to bin the triangles/primitives into intervals between planes. By using the same techniques which have proven successful for kd-tree build techniques, the same techniques proved, in many cases, to be even more efficient for BVHs.

As BVHs can adapt poorly to non-uniformly tessellated scenes, construction algorithms have also been presented to address these issues. Stich et al. [26] presented an approach called the Split Bounding Volume Hierarchy (SBVH) which addresses these issues by utilising spatial splitting similar to kd-trees and using the surface area heurisitc to control primitive splitting during tree construction.

Other top down construction methods include Lauterbach et al.'s [19] morton code builder. The technique "uses a linear ordering derived from spatial morton codes to build hierarchiers extremely quickly and with high parallel scalability".

2.2.2 Bottom-up

Bottom-up methods require the leaves of the tree as the input set. The method starts by grouping two or more of the leaves to form an internal node. Grouping continues in this manner until everything has been grouped and a single node is left, the root of the tree. Bottom-up methods are more difficult to implement than their top-down counterparts but generally produce trees of a higher quality.

Although the preferred approach to BVH construction has been to build the hierarchy top-down, efforts are being made to explore the viability of bottom-up algorithms [33]. Walter et al.'s approach examines the use of agglomerative clustering to build high-quality BVHs. Agglomerative clustering is a greedy algorithm which is initialised with the scene primitives as singleton geometry clusters. The nearest clusters are repeatedly combined until a single cluster is left which represents all the primitives in the scene. Although the technique can produce high-quality hierarchies its use has been limited by its high cost which stems from nearest neighbour computations when joining clusters. Walter et al. attempt to mitigate this cost by accelerating the search by storing remaining clusters in a kd-tree "constructed using a low-cost, top-down partitioning strategy" such as coordinate bisection. This means that a cheap, lower-quality hierarchical acceleration structure is being used to accelerate the construction of the final, higher-quality one. The resulting structure can be of high quality but research of faster clustering techniques is still underway [11].

2.2.3 Insertion

Insertion methods are considered to be "on-line" as they do not require all primtives to be available before construction starts and therefore allow updates to be performed at runtime. The tree is built by inserting one object at a time. The location of the insert is chosen so that the tree grows as little as possible according to some cost metric. On-line methods have been shadowed by the growing success of top-down and botton-up construction algorithms.

2.3 Approaches

As modern processors are becoming increasingly parallel, much of recent work has concentrated on parallel construction of BVHs on both multicore CPUs and GPUs in order to achieve interactive per-frame hierarchy rebuilds. Unfortunately, hierarchy construction does not scale as well as ray tracing with multiple processors. The difficulty lies in the fact that parallelisation is not easy to achieve in the first steps of top-down build procedures. These construction algorithms can also become bandwidth limited since memory bandwidth has traditionally not increased by Moore's law like computing power. Many diverse apporaches to parallelised hierarchy construction have been proposed to overcome these issues. The architecture the structure is being built on can play a major role in its construction.

2.3.1 Construction on Multicore CPUs

Some of the first attempts made at parallel BVH construction were implemented on multicore CPUs. Algorithms were often parallelised by assigning different threads to work on different sub-trees as sub-trees are independant of each other. The drawback of these approaches is that they need to have enough independant sub-trees to work on which is not the case until a number of partitions have been made. In other words, at the root of the tree, only one subtree is available to work on. After splitting the root, only two sub-trees are available. Wald's algorithm strategy to overcome this bottleneck involves mixed horizontal and vertical work sharing [29]. He distinguished between the upper and lower nodes of the tree, utilising a more data parallel approach for the upper nodes and a task parallel per sub-tree scheduling for lower nodes. In addition to construction, parallel refitting techniques for BVHs have been shown on multicore CPUs which help to mitigate BVH quality degradation [18].

More recent work on multicore BVH construction can be seen in the Intel Embree set of ray tracing kernels [5]. The kernels provide a number of build methods which are highly optimised for modern CPUs and demonstrate ray tracing performance competitive with popular GPU approaches.

2.3.2 Construction on Manycore CPUs

Many researchers prefer more high-throughput architectures and so are turning towards GPUs and more recent manycore CPUs like Intel's Many Integrated Core (MIC) architecure. While these architectures benefit from significantly higher compute power which allow them to achieve higher performance they suffer in some other aspects. Their nature of excelling at easy-to-parallelise and compute-intensive tasks like ray tracing and shading hit points leave them off the pace of CPU performance when it comes to more control-intensive tasks such as building complex ray tracing data structures.

Recent parallel BVH construction on manycore CPUs was presented by Wald et al. [31] for the Intel MIC architecure. The MIC architecture consists of 32 x86 cores operating at a frequency of 1GHz. Algorithmically, this implementation resembles earlier work [29] by utilising a data parallel approach for large nodes, and task parallel per sub-tree scheduling for smaller nodes/sub-trees. As addition to this, they use four key concepts to maximise performance: "progressive 10-bit quantization to reduce cache footprint with negligible loss in BVH quality; an AoSoA data layout that allows efficient streaming and SIMD processing; high-performance SIMD kernels for binning and partitioning; and a parallelisation framework with several build-specific optimizations".

2.3.3 Construction on GPUs

High throughput GPUs have also found a place in ray tracing. GPU-based algortithms can achieve build times, of regular data structures like grids, which are just as fast as those of manycore CPU-based approaches. Although CPUs retain an edge when building more complex data structures like BVHs with SAH, feasible SAH and similarly efficient builds have been presented on GPUs.

A breadth-first parallelisation of binned SAH BVH construction has been shown to be effective on GPUs by Lauterbach et al. [19]. A new thread in the build is generated for each child node, allowing for a large number of concurrent threads to effectively utilise the GPU, boosting parallelisation significantly. An alternative hybrid LBVH/SAH scheme was also proposed to extract more parallelism at the top of the tree as the work at the top of the BVH structure is inherently sequential. The authors extended this to the

Hierarchical LBVH, to take greater advantage of data coherence [22]. Further research on the HLBVH has produced faster and more efficient implementations [8, 14].

Another fast approach to binned SAH BVH construction on the GPU was proposed by Sopin et al. [25]. Like other algorithms, nodes are differentiated by their size in order to more efficiently assign tasks to the GPU. Sopin et al.'s approach utilises a larger number of cores for upper nodes and assigns fewer cores per node as the nodes become smaller. The proposed method demonstrated more than tenfold increases in performance compared to other GPU implementations and as such is among the fastest published implementations of the binned SAH BVH construction algorithms.

2.3.4 Construction on Dedicated Acceleration Architectures

Until recently, the aspect of dedicated microarchitectures for the construction of BVHs has been largely absent in research with the majority of the research that does exist on the subject being focused on the traversal and intersection aspects of the ray tracing pipeline. This is unfortunate as efficient and promising strategies have been demonstrated by Doyel et al. [3]. Their custom microarchitecture for the construction of binned SAH BVHs offers many benefits over alternative approaches:

- Acceleration of up to 10x for binned SAH BVHs compared to manycore platforms.
- Low memory bandwidth due to explicit management of on-chip buffers and local register file, and the elimination of instruction fetches. A low memory footprint is also observed.
- Consumption of minimal hardware resources, representing a large efficiency improvement over software BVH builds which typically engage almost the entire chip.
- Indications of power efficiency which are likely to compare favourably to software implementations running on manycore processors.

Aside from these observations, the microarchitecture does carry some disadvantages. The design is fixed-function and also consumes a certain amount of hardware real estate. However, the design is quite configurable in that many parameters of the build can be

changed and could be coupled with programmable cores. The hardware real estate is also comparable to previous traversal hardware.

Chapter 3

Experimental Design and Implementation

This section will cover the high and low-level design of the Approximate Agglomerative Clustering (AAC) algorithm and its implementation into the Intel Embree ray tracing kernels [32]. An overview of the Ebmree renderer, kernels and the AAC algorithm will be given before addressing the actual design and implementation.

To realise the AAC algorithm and investigate its value in an authentic system the Embree ray tracer and kernels were chosen as a development environment. Although Embree was not designed to fulfill the needs of a full featured global illumination renderer it does form a useful basis for research and the kernels can be easily integrated into many arbitrary CPU based ray tracers. The Embree system was adopted in order to undertake the implementation and evaluation of the AAC algorithm.

3.1 Embree Renderer

Embree provides a fully functional path tracer to illustrate the performance of the kernels. The photo-realistic renderer is provided in both scalar and vectorised variations which interface with the kernels through an API (see Figure 3.2). It can be used to demonstrate how Embree can be used in practice and to measure the performance achievable by the kernels in realistic application scenarios.

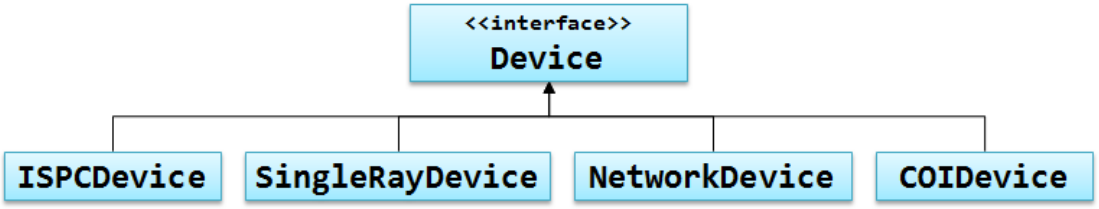


FIGURE 3.1: Renderer/kernel API device interface. The device API ensures that the renderer does not have to worry about where the computation is taking place (Xeon, Xeon Phi, local, remote etc.) [7].

The scalar path tracer is built on top of a set of vector types including vectors, points, colours, which are automatically and transparetly mapped to low-level SIMD types and intrinsics. These vector types allow for the implementation of high-level renderers whose code appears to be scalar but still benefits from vectorization. Besides this, the renderer contains no other vectorization code and rays are traced individually. The full renderer is implemented through C++ and utilises several C++ features such as templates, virtual functions and STL containers.

The path tracer integrator in Embree implements a unidirectional eye path tracer with quasi-monte carlo sampling, russian roulette termination and local evaluation of direct illumination. It also supports ambient, directional, point light sources and a HDRI environment light source with 2D importance sampling. Several non-trivial materials have also been included like brushed metal and multi-layered metallic paint. Materials are composed of mutiple base BRDFs (Bidirectional Reflectance Distribution Function) that can be combined or layered on top of one another.

Although the integrator is simple and does not support advanced techniques such as bidirectional path tracing, photon mapping or multiple importance sampling, it is believed to be sufficiently complex in code design, shading complexity, and ray distributions to be representative of a real-world renderer.

When Embree was designed, almost all real-world renderers were tracing single, individual rays so Embree was also designed for single-ray SIMD kernels. When building for an 8 or even 16 wide SIMD renderer, a purely scalar approach becomes problematic:

- Addition of extra SIMD lanes to a single-ray SIMD kernel produces diminishing returns.

- Even if the single-ray SIMD traversal could be accelerated, Amdahl's law tells us that the overall time to frame could never get faster than the scalar renderer can generate and shade the rays.

To address both issues, the renderer must also be vectorised. This would reduce time spent on shading and enable the renderer to use faster packet/hybrid kernels.

To this end, an additional SPMD-vectorised version of the path tracer was added to the Embree system. This path tracer demonstrates the true performance of Embree's packet/hybrid kernels for incoherent rays.

A "Single Program, Multiple Data" (SPMD) approach was taken to add a vectorised path tracer to the renderer. Implementing fast ray tracers can be difficult and implementations often don't perform optimally on Intel architectures. This is why the Intel SPMD Program Compiler (ISPC) was also chosen for the system which is a compiler for a C-like language with extensions for SPMD programming. The compiler allows for linking, recursion, function pointers and a programming model that mixes both scalar and vectorized data. All shading and sampling is written through the ISPC language and the resulting renderer links to the Embree library where the kernels are accessed through ISPC bindings in the API layer.

3.2 Embree Ray Tracing Kernels

Embree's core consists of a set of low-level, high-performance kernels which are specifically designed for performing two operations: building data structures, and tracing rays. Simultaneously, the tracing of rays is internally split into two types of kernels: traversing an acceleration structure and intersecting primitives. These kernels are templated over a specific primitive representation which means the use of the same traversal kernel is allowed for different primitive types.

3.2.1 Primitives and Data Structures

The kernels operate exclusively on BVHs which allow for fast single-ray queries by testing multiple nodes in parallel. A branching factor of four in the structure is currently the

working standard within the kernels. In particular, a quad BVH (BVH4) is employed which allows both packet and single-ray kernels to concurrently operate on the same data structure.

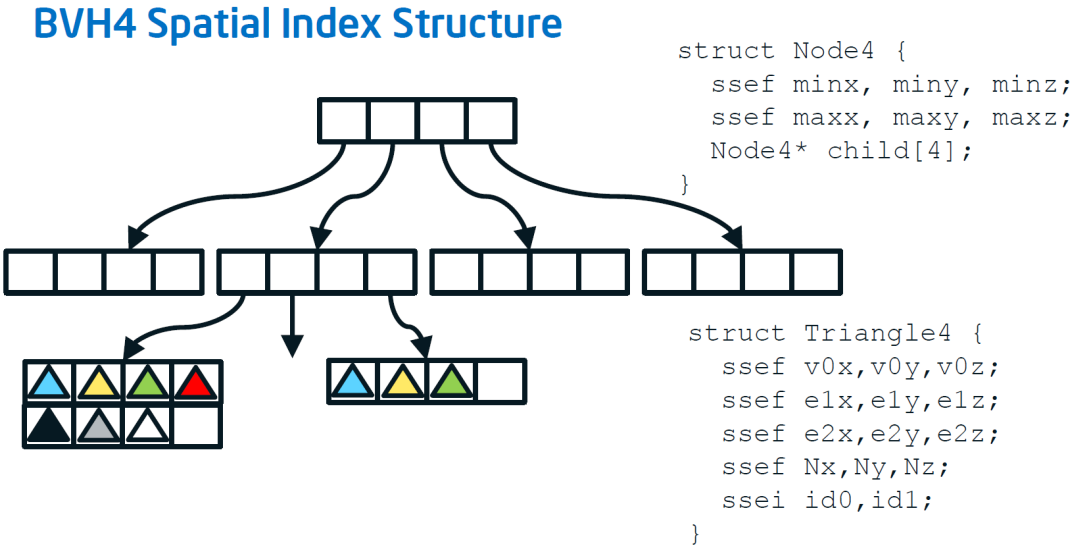


FIGURE 3.2: Quad BVH (BVH4). Each node of the structure contains four children which point to another node or a leaf (e.g. Triangle4 primitive) [35].

Embree also abstracts the definition of a "primitive" from its implementation. This enables the construction of a BVH in which the leaf nodes contain one or more triangles defined as vertex indices or pre-gathered/pre-computed vertex data. Structures like the bvh4i store indices to primitives in their leaves while the bvh4 stores pointers. For a triangle{1,4,8}{i,v,n} primitive, the first set set of braces represent either 1, 4, or 8 triangles in a SIMD friendly layout while the second set of braces represents whether the primtive is a record that stores either the vertex indices (i), the actual vertices (v), or some preprocessed edge and normal data (n).

3.2.2 Traversal and Intersection Kernels

To keep the number of kernels required for traversal and intersection to a minimum, many features (different BVH and primitive representations, ISAs, intersection tests, packet vs single-ray queries) are handled through compile-time flags.

For single-ray SIMD triangle intersection on 4-wide SSE-based architectures, both triangle1n and triangle4n layouts are supported. The kernels use an improved variant of the Möller-Trumbore test [16] for intersection. With the triangle1n layout, SSE is used to process x, y, and z in parallel in the same SSE vector. For triangle4ns, each SSE vector lane processes a different triangles intersection test and then, a final reduction is performed to determine which potential hit is the closest. This is generaly faster then the triangle1n approach.

For 8-wide AVX/AVX2-based architectures, triangle8ns are supported which can intersect 8 triangles in parallel. As AVX and AVX2 are also backwards compatible to SSE, the existing triangle4n variants on AVX can still be used.

On 16-wide Xeon Phis, operations take place on four triangles in parallel using algorithms described by Benthin et al. [2]. 16-wide SIMD registers of the Xeon Phi are treated as if they were four parallel 4-wide SIMD units, and processes four different triangles' (x, y, z) tuples in parallel. As the Xeon Phi is powerful enough to work on four different triangle1ns for intersection, a special triangle4n data layout is not needed.

For single-ray BVH traversal on architectures with Intel Streaming SIMD extensions 4.1 (SSE 4.1), four child nodes are always tested in parallel. The BVH4 node layout is designed to efficiently support this and exploit available parallelism.

On architectures with Intel Advanced Vector Instructions (AVX/AVX2), there is no easy way of exploting 8-wide SIMD when testing four boxes so instead, the kernels use the same 4-node algorithm as for SSE. The triangle8n representation and new AVX2 instructions like the fused multiply add are also exploited.

An improved version of Benthin et al.'s approach is implemented for single-ray BVH traversal on the Xeon Phi. The flexibility of the Xeon Phi ISA is exploited to perform four box tests with x, y, and z in parallel [2].

Embree's packet kernels are conceptually trivial, compared to the single-ray kernels which require careful mapping of computations to SIMD lanes. The packet kernels have all lanes operate on the same triangles respectively while quad-nodes allow for more efficient control flow as well as spreading of computations out across SIMD lanes. There

are different implemenations for each of the ISA's different SIMD widths and kernels in 4, 8, and 16-wide variations.

Hybrid single-ray and packet traversal can be accomplished when using the same data structure for both packet and single ray queries. This allows for traversal routines to use packets until a case where rays become too incoherent for packet tracing, in which case, traversal falls back to single-ray traversal. The hybrid single-ray/packet scheme ensures the renderer will perform quickly for incoherent rays and fall back to single-ray kernels for incoherent work even when used by vectorised renderers.

3.2.3 BVH Construction Kernels

Embree exploits a number of different build procedures which are utilised under different circumstances based on mesh complexity or thread availability for example. Embree differentiates between high-quality and high-performance builders although all builders are parallelised, which means that even the high-quality builders are quite fast.

High quality BVH builds are achieved through a combination of object partitioning and spatial splits [26]. The partitioning is accomplished by SAH binning [29]. At the top of the tree, a max of 32 bins are used. This number is gradually decreased as the build moves further down the tree. For the spatial splits, only one plane is tested at the spatial centre of each dimension. This reduces the SAH quality but is significantly faster compared to multiple split based approaches [26]. This method also still allows for splitting the long, diagonal triangles which are the main source of problems for non spatial split based BVH builders.

Fast BVH builds operate with the SAH binning exclusively and manage triangles using ranges inside a single array. Binning and partitioning are performed by all threads in parallel. Once the size of a partitioning task drops below a threshold, it is handled by a single thread alike to Wald et al. [29].

Embree also implements a high-performance morton-code builder following Lauterbach et al. [19]. The builder exploits task parallelism by using all threads to cooperatively compute centroid bounding boxes and calculate the list of morton codes which are stored as 64-bit key/value pairs. Once the morton codes are determined, they are sorted using

Embree's own radix sort procedure. The construction of the BVH structure is then carried out by searching for differences in the morton bits. The construction begins by having all threads build entire levels of the tree. When enough sub-trees are available, the second phase begins in which a single thread is assigned per sub-tree. The builder creates a leaf node as soon as a sub-tree has less than 4 triangles.

3.2.4 Embree Kernel API

The Embree API offers users a thin and straightforward interface for using the kernels. The low-level nature of the API allows it to support a host of features:

- Defining and committing geometry.
- Building acceleration structures over geometries and defining their type options e.g. performance vs compactness.
- Performing ray queries and defining their type e.g. single-ray vs 4, 8, and 16 wide packets.
- Static and dynamic scenes.
- Motion blur.

The API abstracts the majority of technical details from the user which means high performance and fast ray tracers can be built without needing a deep knowledge of the kernels. The API is also implemented in both C++ and ISPC to allow for the use of the vectorised version of the ray tracer. The ISPC version is almost identical to the C++ version with the exception of ISPC specific uniform type modifiers and limiting of ray packets to native SIMD sizes which the ISPC code is compiled for.

The main data type provided by the API core is the "RTCScene". RTCScene is a "container for a set of geomtries of potentially different types" [12]. A new scene is created by calling the "rtcNewScene" function and destroyed by calling "rtcDeleteScene". Once a call has been made to create a scene, Embree internally chooses the best combination of data structures and kernels to use based on condition flags which are provided at scene creation. These flags specify the type of scene that can be created (e.g. static, dynamic)

and the type of ray query operations that can be performed on the scene. For static scenes, a single BVH4 is built over the entire scene and constituent meshes and the primitive layout is based on compactness and coherence flags specified by the user. Dynamic scenes are treated differently and involve the implementation of a two-level hierarchy. First, a BVH4 is built for each individual mesh in the scene. Then, a top-level BVH4 is built over these BVHs. The builder used for each stage depends on the specified flags. Each mesh uses either standard binned SAH for static meshes, the refitting kernel for deformable meshes, or the morton code builder for truly dynamic meshes. The top-level builder constructs the over-arching tree using each meshes' BVH4 root nodes.

3.3 Approximate Agglomerative Clustering

Agglomerative clustering is a greedy algorithm which is initialised with the scene primitives as singleton geometry clusters. The nearest clusters are repeatedly combined until a single cluster is left which represents all the primitives in the scene. Although the technique can produce high-quality hierarchies, its use has been limited by its high cost which stems from nearest neighbour computations when joining clusters. Walter et al. attempt to mitigate this cost by accelerating the search by storing remaining clusters in a kd-tree "constructed using a low-cost, top-down partitioning strategy" such as coordinate bisection. This means that a cheap, lower-quality hierarchical acceleration structure is being used to accelerate the construction of the final, higher-quality one. The resulting structure can be of high quality but research of faster clustering techniques like AAC has been underway [11].

Approximate Agglomerative Clustering has been introduced by researchers Gu et al. [11] as an efficient and easily parallelisable algorithm for generating high-quality bounding volume hierarchies. The main idea of the AAC algorithm is to "compute an approximation to the true greedy agglomerative clustering solution by restricting the set of candidates inspected when identifying neighbouring geometry in the scene". Gu et al. report that the structures produced by the AAC algorithm are often of comparable or higher quality than those of top-down, full sweep SAH builds and can be constructed in less time than the widely used fast SAH builds with "binning" [29].

The AAC algorithm was developed as a result of interpreting the weaknesses of locally-ordered or heap-based agglomerative clustering [33]. These techniques suffer from expensive computation costs at the beginning of the construction process when the number of clusters to combine is very large. This weakness is a result of initialising one cluster per scene primitive which means, at the bottom of the tree, each nearest neighbour search requires a global operation over all primitives "incurring cost at least $O(logN)$". This cost represents a large portion of the total computation and operations to reduce this cost would speed up the overall procedure exponentially.

Gu et al.'s approach stems from this idea of reducing the computational cost of finding a cluster's nearest neighbour. The algorithm restricts the nearest neighbour search to small subsets of neighbouring scene elements/clusters, reducing the number of elements which need to be tested on each search. Gu et al. shows that "this approximation minimally impacts resulting BVH quality but significantly accelerates the speed of BVH construction".

The main concept of AAC is to quickly organise the scene primitives into a binary tree based on recursive coordinate bisection. The leaf nodes of the resulting tree (the constraint tree) each contain a small set of scene primitives (singleton clusters) which are relatively close in proximity (see Figure 3.3). BVH construction via agglomerative clustering then proceeds up with the constraint tree controlling which primitives can be clustered together. At each node of the tree, a set of un-combined clusters is generated by taking the union of the set of un-combined clusters of its children. This set is then greedily reduced and passed up the tree. This continues until the root of the constraint tree has been reached and the clusters are reduced to a single cluster.

The AAC method follows three distinct phases which include a sorting phase, a "downward phase", and an "upward phase".

In the sorting phase, Morton codes are first generated for the scene primitives in relation to their bounding box centres. The Morton curve, also known as a z-order curve is used for ordering primitives. The binary fixed-point representation of each coordinate is taken and expanded and interleaved to form a single binary number. The leaf nodes of the tree can then be determined by assigning Morton codes to each object in the scene (see Figure 3.4).

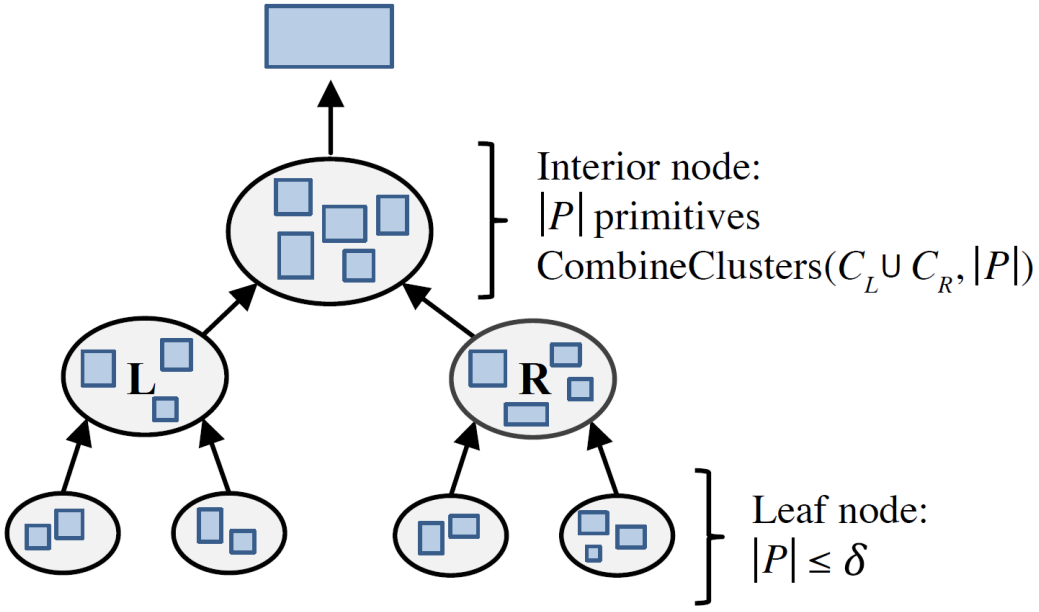


FIGURE 3.3: AAC Constraint Tree. "Constraint Tree" with δ primitives at the leaves. At each node, clusters from that node's children get combined through agglomerative clustering. At the top of the tree, all clusters have been combined into one cluster which represents the BVH root node. [11].

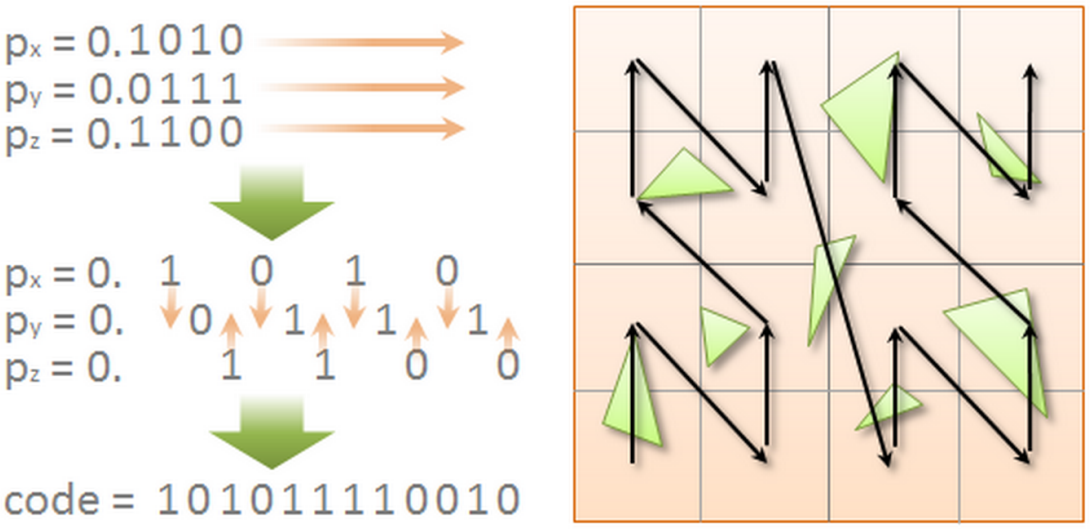


FIGURE 3.4: Morton Curve and Code Construction. Binary fixed-point representation of coordinate is taken and expanded and interleaved to form a single binary number. A Morton curve of the representative points in the scene can be found on the right [15].

The primitives are sorted by the differences contained in their Morton codes. Gu et al. prefer a variable-bit raix sort for the operation which is easily parallelised and cheap to compute. Once the primitives are sorted by their Morton codes they are then used to construct the BVH in the next two phases (see Algorithm 1).

Algorithm 1 AAC(P)

Input: list of all scene primitives P
Output: BVH containing primitives in P

1: Compute Morton code for centers of P;
2: RadixSort P by Morton code;
3: C = BuildTree(P);
4: **return** CombineClusters(C, 1);

The "downward phase" of the algorithm comprises of constraint tree traversal. The constraint tree, implicit in the morton codes, is traversed recursively through the function *BuildTree*. The *BuildTree* function organises the primitives for clustering as it descends the constraint tree. At each node or traversal step, the current spatial extent is split based on the next bit in the morton code (Algorithm 2, line 6). The *BuildTree* function is then called on these new child nodes and the phase continues in this way until the termination criterion has been met (Algorithm 2, line 1).

Algorithm 2 BuildTree(P)

Input: subset of scene primitives P
Output: at most $f(|P)$ clusters containing primitives in P

1: **if** $(|P| < \delta)$ **then**
2: Initialise C with P;
3: **return** CombineClusters(C, $f(\delta)$);
4: **end if**
5: $\langle P_L, P_R \rangle$ = MakePartition(P);
6: C = BuildTree(P_L) $\cup$ BuildTree(P_R);
7: **return** CombineClusters(C, $f(|P|)$)

The final "upward phase" involves the actual bottom-up clustering (see Algorithm 3). This stage begins once the number of primitives in clusters in the downware phase has dropped below the threshold δ. Once this has occurred, the *CombineClusters* function begins the work of clustering primitives as it traverses back up the tree organised by the *BuildTree* function. Each call to *CombineClusters* combines the clusters returned from the left and right constraint tree nodes. This recombination phase continues up

the tree until one cluster remains. This reduction of clusters results in the root of the BVH which contains all primitives in the scene.

Algorithm 3 CombineClusters(C, n)

Input: list of clusters C
Output: list of at most n clusters

```
1:  for all C_i ∈ C do
2:      C_i.closest = C.FindBestMatch(C_i);
3:  end for
4:  while |C| > n do
5:      Best = ∞;
6:      for all C_i ∈ C do
7:          if d(C_i, C_i.closest) < Best then
8:              Best = d(C_i, C_i.closest;
9:              Left = C_i; Right = C_i.closest;
10:         end if
11:     end for
12:     c' = new Cluster(Left, Right);
13:     C = C − {Left, Right} + {c'};
14:     c'.closest = C.FindBestMatch(C_i);
15:     for all C_i ∈ C do
16:         if C_i.closest ∈ {Left, Right} then
17:             C_i.closest = C.FindBestMatch(C_i);
18:         end if
19:     end for
20: end while
21: return C;
```

Once the root of the constraint tree has been reached, the clusters are reduced to a single cluster (see Algorithm 1, line 4).

The parameters of the AAC algorithm also play a significant role in the speed of construction and quality of the BVH. The two parameters consist of the "cluster count reduction fuction f" and the "traversal stopping threshold δ". These parameters affect how much of the agglomerative clustering is approximated, which in turn affects the construction cost and the BVH quality.

3.4 Implementation

The implemenation of the AAC algorithm meant delving into the innards of Embree's construction kernels. The kernels use many different classes with quite a bit of abstraction which first had to be discerned. The numerous code paths for using different construction and intersection kernels controlled by the compile time flags also had to be clearly understood.

Before beginning the work, some setup had to performed to get Embree up and running. First, the Intel SPMD Program Compiler (ISPC) had to be installed which is used to compile the vectorised version of the Embree pathtracer. Once ISPC was correctly installed on the machine, the Kernels and Embree renderer had to be installed and correctly configured for use which involved some tedious library linking and project properties arrangement through both the System variables and Visual Studio, the IDE.

Once the working environment for Embree was up and running and a working knowledge of the kernels was achieved, implementation work could begin.

The Morton code builder of the kernels was the most suitable place to begin development as some of the functionality required for the AAC algorithm was already present. To test the AAC algorithm with different scenes during development, the code path selection was restricted to lead the kernels to use the AAC builder for construction of BVHs.

An *AAC_build* function was implemented which can be called instead of the Morton code build. All three of the AAC stages occur in this function.

For the first phase of the AAC algorithm, the *computeMortonCodes* and *radixSort* functionality was exploited which was already in place for the pure Morton builder in the kernels.

The second phase of the algorithm involves travseral of the constraint tree (implicit in the Morton codes) and splitting of primitives and nodes. This is where the implementation for AAC diverges from the kernel code. In the other Embree builders, primitives are represented by the *SmallBuildRecord* class which stores the start and end of a set of primitives, the depth of this record as a node, and a reference to its parent node. The use of Embree data types is exploited by passing an initial *SmallBuildRecord* to the

main build function, *BuildTree*, with all scene primitives similar to the pure Morton code builder. The beginning of the downward phase can be seen in Listing 3.1.

```
// Pass initial build record to BuildTree()
SmallBuildRecord br;
br.init(0, numPrimitives);
br.parent = &bvh->root;
br.depth = 1;

nodeAllocator.reset();
primAllocator.reset();
__aligned(64) Allocator nodeAlloc(nodeAllocator);
__aligned(64) Allocator leafAlloc(primAllocator);

BuildTree(br, nodeAlloc, leafAlloc, RECURSE, threadIndex);
```

Listing 3.1: Beginning of Downward Phase

Once the *BuildTree* function begins, the input build record primitive count is checked against the traversal threshold. If the primitive count is above the threshold, the build record is split into four smaller build records/children which are assigned the current build record as their parent node. Each of these children are then passed as parameters to a further *BuildTree* call as can be seen in Listing 3.2.

```
// Primitives and Nodes are split
// Each child is passed to another BuildTree() call
for (i = 0; i < numChildren; i++){
    children[i].parent = &node->child(i);
    children[i].depth = P.depth + 1;

    Clusters[i] = BuildTree(children[i], nodeAlloc, leafAlloc, mode, threadID);

    combine(C, Clusters[i]);
    Clusters[i].depth = P.depth + 1;
    node->set(i, Clusters[i].bounds);
    Clusters[i].parent = &node->child(i);
}
```

Listing 3.2: Primitive and Node Splitting

The algorithm continues in this fashion until the traversal threshold has been reached. At that point, a *Cluster* class is instantiated and initialised with the input primitives and

their bounds. The *Cluster* class is an implementation of the AAC cluster structure which contains the same attributes as a *SmallBuildRecord* with the exception of its vector of *SmallBuildRecords*, *SBRClusters*, representing the clusters in the structure. The *Cluster* class also has a number of functions related to it such as *FindBestMatch* which is used for calculating a cluster's nearest neighbour by examining the Morton codes of clusters, or the *combine* function which takes the union of two clusters.

In the third stage of the algorithm, the upward clustering begins. At the bottom of the tree, or when the splitting threshold has been reached through stage two, a cluster is initialised with those primitives and passed to the *CombineClusters* function as can be seen in Listing 3.3.

```
// Cluster C at bottom of tree initialised with primitives in P
// C passed to CombineClusters()
Cluster C;
C.bounds = empty;
if (P.size() <= TRAVERSAL_THRESHOLD){
    C.SBRClusters.resize(P.size());
    unsigned int end = P.begin + P.size();
    int j = 0;
    for (size_t i = P.begin; i < end; i++){
        C.SBRClusters[j] = P;
        C.SBRClusters[j].begin = i;
        C.SBRClusters[j].end = i + 1;
        C.SBRClusters[j].depth = P.depth;
        j++;
    }
    C.parent = P.parent;
    return CombineClusters(TRAVERSAL_THRESHOLD, C, nodeAlloc, leafAlloc, 1, ↩
    threadID);
}
```

Listing 3.3: Cluster Initialisation and Beginning of Bottom-up Clustering

In the *Combine Clusters* function, the clusters are paired with their nearest neighbour using the *FindBestMatch* function which examines the Morton codes of the clusters' primitives for changes. Once each cluster's nearest neighbour is calculated, the clusters are combined together until n clusters remain (n represents the clustering threshold as opposed to the splitting threshold *delta* used in the downward phase of the algorithm). The clusters are combined by choosing the pair of clusters with the shortest distance

between them and combining them together. The list of cluster's nearest neighbours are then recalculated after the change. This procedure of cluster combination continues until a threshold n has been reached. When this happens the cluster is returned. The clusters returned from this function are combined using the *Cluster* class' *combine* function and this cluster's bounds are then assigned to the current node. This procedure continues until all clusters have been combined and only a single cluster remains, representing the root node of the tree.

Figure 3.5 represents a graphical interpretation of the AAC implementation in the kernels. The image describes the splitting of the nodes and primitives in the second phase of the algorithm and the synthesis of the node bounds and recombination of clusters on the way back up the tree. The blue areas of the derivation tree represent the downward phase while the red areas represent the upward phase. The input to the downward phase is the inital *SmallBuildRecord* containing the scene's Morton ordered scene primitives and BVH root node. The upward phase produces the finished BVH structure as a result of the recursive clustering and bounds assignment to the internal nodes of the BVH.

The initial implementation of the AAC algorithm was quite slow and a number of areas of the code had to be debugged in order for a correct hierarchy to be built for the kernels. A modified version of *refit_toplevel* function in the Morton builder was used to run over the constructed tree to calculate correct bounds and test if at least the tree structure was correct. Once the program was producing correct trees, some simple optimisations were made to make the build times more acceptable. Following this, the implementation was further examined and some initial evaluations of the hierarchies were taken in order to improve the quality of the trees produced and the performance of their construction.

3.4.1 Parallelisation

The first implementation of the AAC algorithm was performed in a single-threaded fashion. Once the correct trees were being produced and at a reasonable pace, an attempt at parallelising the procedure was made. In similar fashion to the Morton builder, parallelisation was found by assigning different threads to work different sub-trees as sub-trees are independant of each other. This occurs in the downward phase

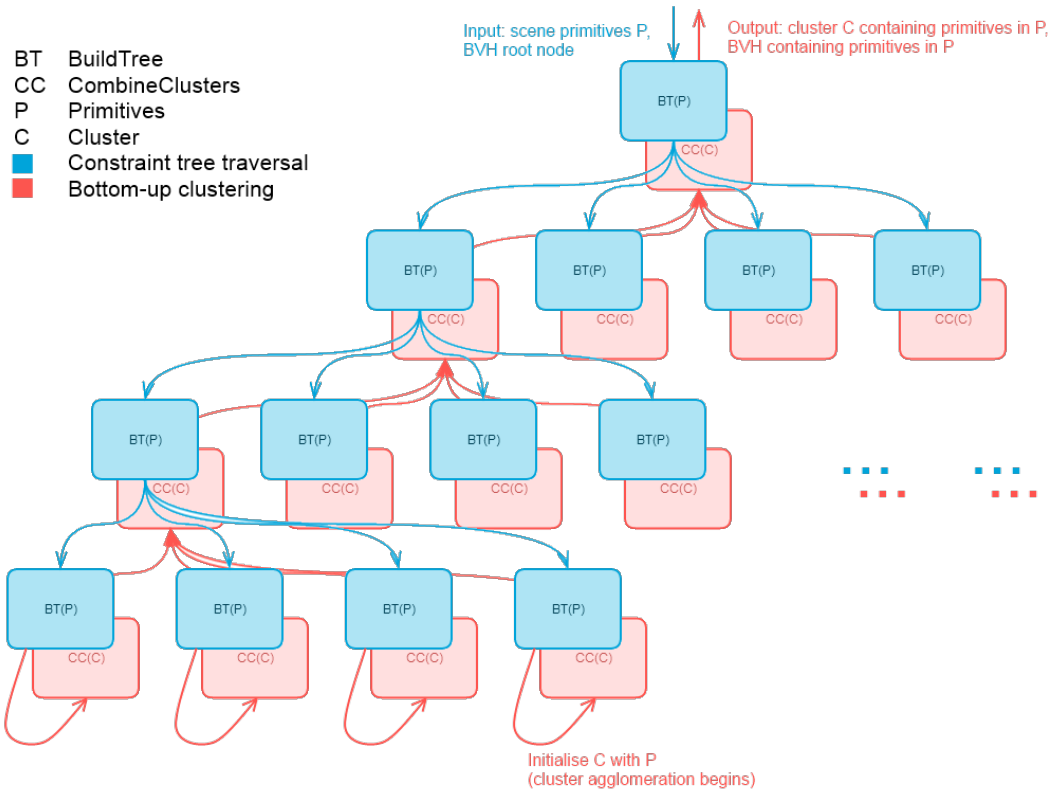


FIGURE 3.5: AAC Derivation Tree. The scene primitives and BVH root node are passed in a *SmallBuildRecord* as input to the algorithm and the downward phases (blue). The clusters are initialised with primitives at the bottom of the tree and are passed as input to the upward phase of the algorithm (red) to be clustered, resulting in the final BVH at the top of the tree as output.

once the number of subtrees available has grown sufficiently large to keep all threads busy.

At the root of the tree, only one sub-tree is available to work on. After splitting this sub-tree, only two, or four in this implementation's case, are available to work on. A single thread continues to work on the downward phase until a sufficient amount of sub-trees are created to work on.

Before parallelisation, the *BuildTree* function was called from the *build_sequential_AAC* function. For parallelisation, a new *build_parallel_AAC* function was introduced. *BuildTree* is also called in this implementation but the parameters are slightly different. A parameter representing the mode type of the build procedure is passed the enum variable *CREATE_TOP_LEVEL* which signifies that the *BuildTree* should be building the "top level" of the BVH tree only. The *BuildTree* function was also modified for parallelisation.

At the start of the method, the mode parameter is checked. If the mode is to create the top level of the tree and the number of primitives in the input P is less than a top level item threshold, P (representing a subset of primitives) is assigned to an index in an array of build records in the builder "state" class. Once the top level has been created, the *BuildTree* function returns to *build_parallel_AAC*. The new task *task_buildSubAACTrees* is then dispatched to the thread scheduler of the kernels. In *buildSubAACTrees*, the SmallBuildRecords (primitive subsets) assigned to the build record array previously are passed to the *BuildTree* function again, except this time, with a mode set to the enum variable *RECURSE*. A thread is assigned to each of these build procedures so as to construct the sub-trees in parallel (see Listing 3.4.

```
void BVH4BuilderMorton::buildSubAACTrees(const size_t threadID, const size_t ↩
    numThreads)
{
    __aligned(64) Allocator nodeAlloc(nodeAllocator);
    __aligned(64) Allocator leafAlloc(primAllocator);
    while (true)
    {
        const unsigned int taskID = scheduler.taskCounter.inc();
        if (taskID >= g_state->numBuildRecords) break;
        BuildTree(g_state->buildRecords[taskID], nodeAlloc, leafAlloc, RECURSE, ↩
    threadID);
        g_state->buildRecords[taskID].parent->setBarrier();
        g_state->workStack.push(g_state->buildRecords[taskID]);
    }
}
```

Listing 3.4: Sub-tree Thread Assignment.

Once the tree is constructed, the top level of the tree structure is refitted using the Morton builder's refit method and the threads are released.

3.4.2 Integration with the Kernels

In terms of the actual AAC builder, the biggest problem was to get it cleanly integrated with the existing build infrastructure, because that infrastructure wasn't designed for

agglomerative clustering. There also isn't any explicit "scalar-only" code path for building in the kernels at the time of writing. As a result, it took some time to establish how the builders were working.

3.4.3 Kernel Features Utilisation

A great advantage of implementing AAC in the Embree kernels is that a number of features and CPU utilisation techniques that the kernels have to offer could be exploited.

One major utilisation of Embree features is the exploitation of the BVH4 structure. By splitting the primitives and nodes of the tree into four, a quad BVH structure is used which allows the hybrid packet traversal kernels to be used, meaning both packet and single-ray kernels can concurrently operate on the same structure which optimises performance in cases where rays may become too incoherent for packet tracing.

By exploiting Embree's leaf creation functionality the leaves of the hierarchy can also employ the kernels primitive abstractions, Triangle{1/4/8}{i/v/n}, which allow for more efficient SIMD utilisation.

Parallelised and optimised approaches for constructing and sorting Morton codes in the Morton code builder kernel were also utilised. The implementations of the Morton code construction and sort algorithms provided in the kernels are highly optimised for x86 CPUs and thread parallelism and as such, provided a great starting point for the implementation of the AAC algorithm.

Chapter 4

Evaluation

All results are computed on a 2.80GHz Intel Core i7-4900MQ CPU with 4 cores and 8 threads using the vectorised Embree pathtracer compiled through ISPC.

All implemented code was written in C++ in the Embree kernels. Exhaustive optimsations were not performed (e.g. alternative to vectors for cluster grouping) and as such build times for the AAC implementation are not fully representative of actual achievable times. Fast build times for AAC have been documented by Gu et al. [11] Although build times are not as fast as possible, they are sufficient enough for comparing the relative merits of the different BVH building strategies.

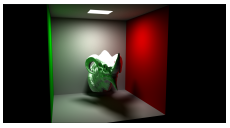
The Embree kernels provide some statistical analysis functionality for analysing BVH hierarchy built times, primitive count, vertices count, SAH, leaf SAH, tree depth and memory consumption statistics. These features were employed to compare the AAC algorithm against Embree's SAH builder, Spatial Splits builder, Morton builder and Refit builder. The algorithms are compared to one another in a number of different scenes with different mesh complexities.

At the time of writing, Embree does not support features for using build times as a precursor to acceleration structure builder choice. As a result, tests are not very well suited for animated scenes as the type of builder is currently chosen based on whether a mesh is static, dynamic or deformable alone e.g. if the mesh is deformable, Embree will use the Refit kernels; if the mesh is dynamic the Morton builder will be used; if the mesh is static a binned SAH or Spatial Splits approach will be taken.

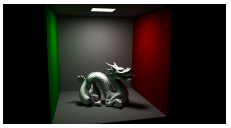
Model	No. Primitives
Gold Sphere	19802
Dragon Bust	422820
Dragon	871322
Demon	935252
Buddha	1087467
Woola	1281039
Imperial Crown of Austria	4868924
Dragon Bust B	8704016

TABLE 4.1: Model Complexity

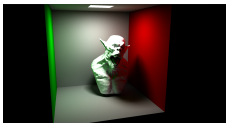
(A) Gold Sphere



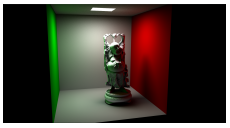
(B) Dragon Bust



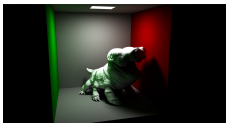
(C) Dragon



(D) Demon

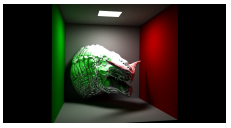


(E) Buddha



(F) Woola

(G) Crown



(H) Dragon Bust B

To exploit Embree's statistical measurements functionality "-rtcore verbose = 2" is passed at the command line, and the Embree builder prints the surface area heuristic cost and other useful information after construction. These costs and statistics are calculated in the *bvh4_statistics.cpp file*.

Drawing the generated bounding boxes is not supported, however the Embree tutorials support some cost visualization modes where the number of clock cycles for each pixel are visualized which can be enlightening.

The models chosen for the evaluations range from quite simple meshes to complicated ones with primitive counts of up to nine million. The Imperial Crown of Austria is available on the Embree website courtesy of Martin Lubisch (http://www.loramel.net). There are a number of scenes included with the Embree package, one of which is the Gold Sphere scene which was used for evaluation. The models "Dragon" and "Buddha" were acquired from http://graphics.cs.williams.edu/data/meshes.xml while the rest of the models were taken from http://www.turbosquid.com/ (Dragon Bust, Demon, Woola, Dragon Bust B). The primitive counts for each model used can be seen in Table 4.1 as a measure of complexity.

As the imperial crown was the only model shipped with the kernels to contain a scene, the cornell box scene (included with Embree) is used as a container for the rest of the models for examining both correct lighting of models and correctness of form as a result of their BVH acceleration structures. The Imperial Crown of Austria and its scene can be seen in Figure 4.2. The Dragon Bust B model can be seen in the cornell box scene in Figure 4.3.

FIGURE 4.2: Imperial Crown of Austria & Scene Render. BVH constructed using AAC algorithm.

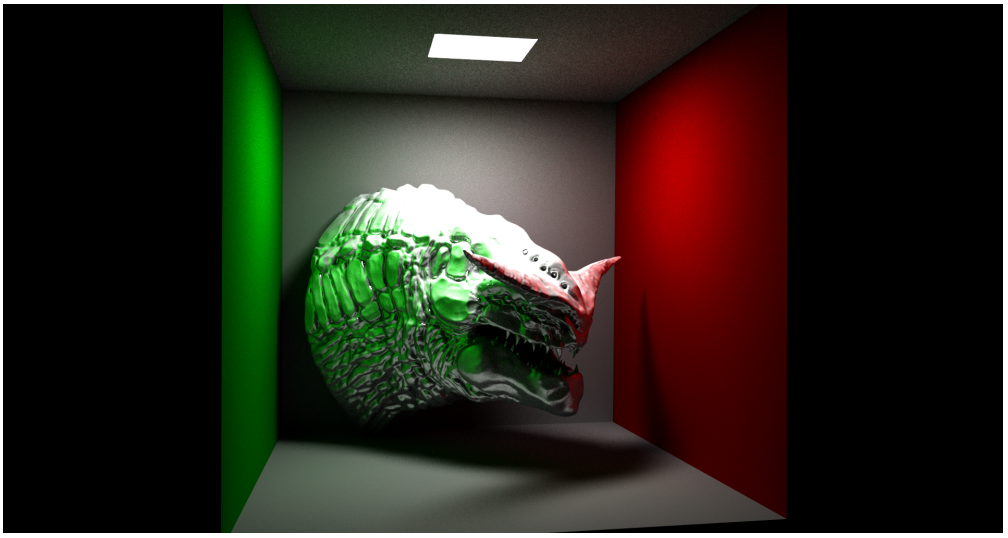


FIGURE 4.3: Dragon Bust B & Cornell Box Scene Render. BVH constructed using AAC algorithm.

Running the Embree pathtracer through the command line, the verbose setting is passed in order to print out BVH construction information. The pathtracer was run with each model and with each builder in the kernels for a complete comparison to be put together. The results of hierarchy construction performance and structure quality are presented in the following sections.

4.1 BVH Build Times

The first evaluations which were made involved looking at the build performance between the sequential AAC implementation and the parallelised implementation. Figure 4.4 shows a graph of build times in milliseconds plotted against the scene primitive counts in relation to both implementations of AAC. This allows us to see what impact the parallelisation of the implementaiton had to the construction of the hierarchies.

The rest of the evaluation work excludes the sequential implementation of the AAC algorithm and solely compares the parallelised version to the rest of the Embree builders. A comparison of build performance can be see in Figure 4.5 which once again plots build times in milliseconds against scene primitive counts in relation to the different build procedures.

4.2 BVH Quality

The Surface Area Heuristic (SAH) is widely used as a predictor for ray tracing performance as it gives the expected cost of tracing a ray through the scene.

Using the SAH, the expected number of box and triangle intersections for a random line is computed based on the surface areas of the clusters in the BVH tree. SAH measurements on the constructed BVH for each of the algorithms and scenes are shown in Figure 4.6.

As well as calculating the BVH structures overall SAH, Embree provides analysis of hierarchy leaf SAH values also. Leaf quality information can be seen for each of the algorithms and scenes in figure 4.7.

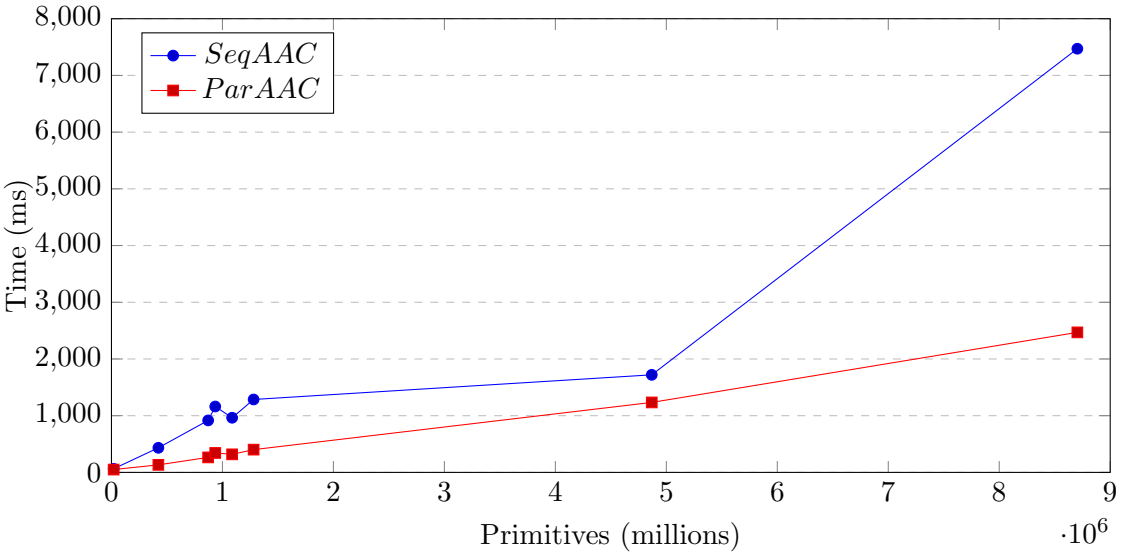


FIGURE 4.4: BVH Build Times (AAC)

Model	*SeqAAC*	*ParAAC*
Gold Sphere	63.5ms	49.8ms
Dragon Bust	433.6ms	133.1ms
Dragon	916.7ms	264.4ms
Demon	1161.9ms	342.7ms
Buddha	963.6ms	318.4ms
Woola	1285.9ms	401.4ms
Imperial Crown of Austria	1720.8ms	1233.8ms
Dragon Bust B	7470.9ms	2469.0ms

TABLE 4.2: BVH Build Times (AAC)

4.3 Discussion

The goal of this paper was to examine the effectiveness and merits of the Approximate Agglomerative Clustering algorithm in relation to more established BVH construction techniques. These tests and figures recorded provide a good point at which the different algorithms can be compared.

The build performance differences between the sequential AAC implementation and the parallelised version were first looked at to measure the impact that the parallelisation had. The effect of splitting the BVH construction work up by assigning threads to different sub-trees of the hierarchy had a significant effect on build performance with the parallelised implementation showing more than a 60% decrease in build time for the Dragon Bust B model and the growth of the graph suggests that gap would increase

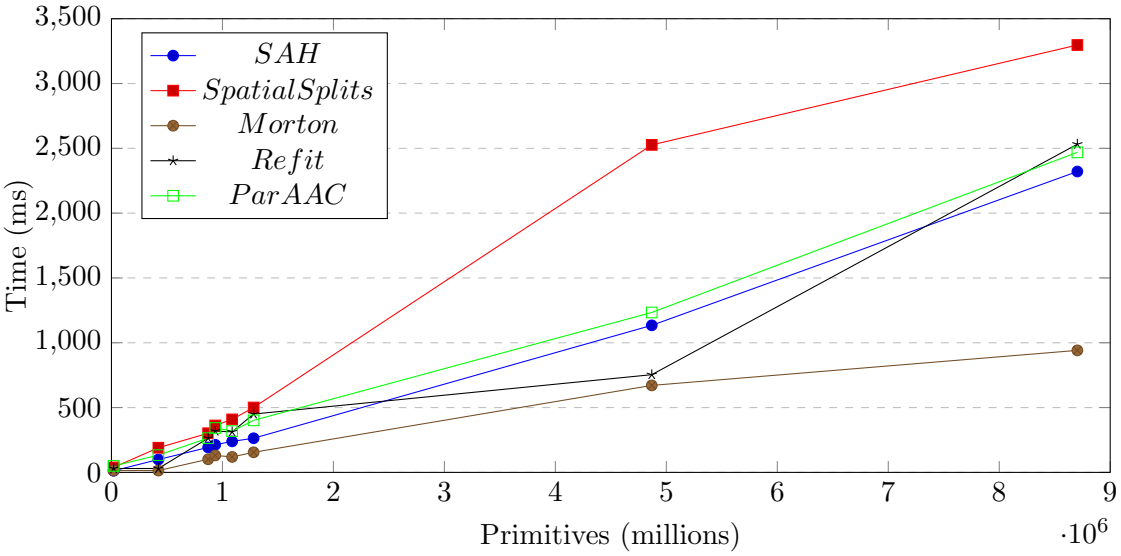


FIGURE 4.5: BVH Build Times

Model	SAH	*SpatialSplits*	*Morton*	*Refit*	*ParAAC*
Gold Sphere	12.5ms	28.4ms	14.1ms	29.9ms	49.8ms
Dragon Bust	99.3ms	190.1ms	82.6ms	137.4	133.1ms
Dragon	192.6ms	302.1ms	101.2ms	263.6ms	264.4ms
Demon	213.1ms	361.9ms	130.8ms	318.3ms	342.7ms
Buddha	239.1ms	408.8ms	119.2ms	312.5ms	318.4ms
Woola	263.4ms	500.6ms	155.0ms	450.1ms	401.4ms
Imperial Crown of Austria	1134.4ms	2526.5ms	671.0ms	753.7ms	1233.8ms
Dragon Bust B	2321.1ms	3297.1ms	941.3ms	2531.8ms	2469.0ms

TABLE 4.3: BVH Build Times

with further model complexity. Even at relatively low poly counts, the overhead for handling the threads was worth the speedup achieved as the parallelised version beat the sequential implementation in all cases. For the rest of the tests the parallelised AAC algorithm was used to compare against the other builders.

The AAC BVH construction performance was evaluated by comparing its build times against that of the alternate builders (binned SAH, spatial spltis, morton, refit). Each of the builder kernels have undergone substantial optimisation and are well knitted with the rest of the kernels and their features, unlike our implementation of the AAC algorithm which is quite simple in its implementation and is relatively unoptimised. Considering this, the AAC algorithm's performance results show it to be just as fast as the binned SAH approach and this would suggest that it would be significanlty faster

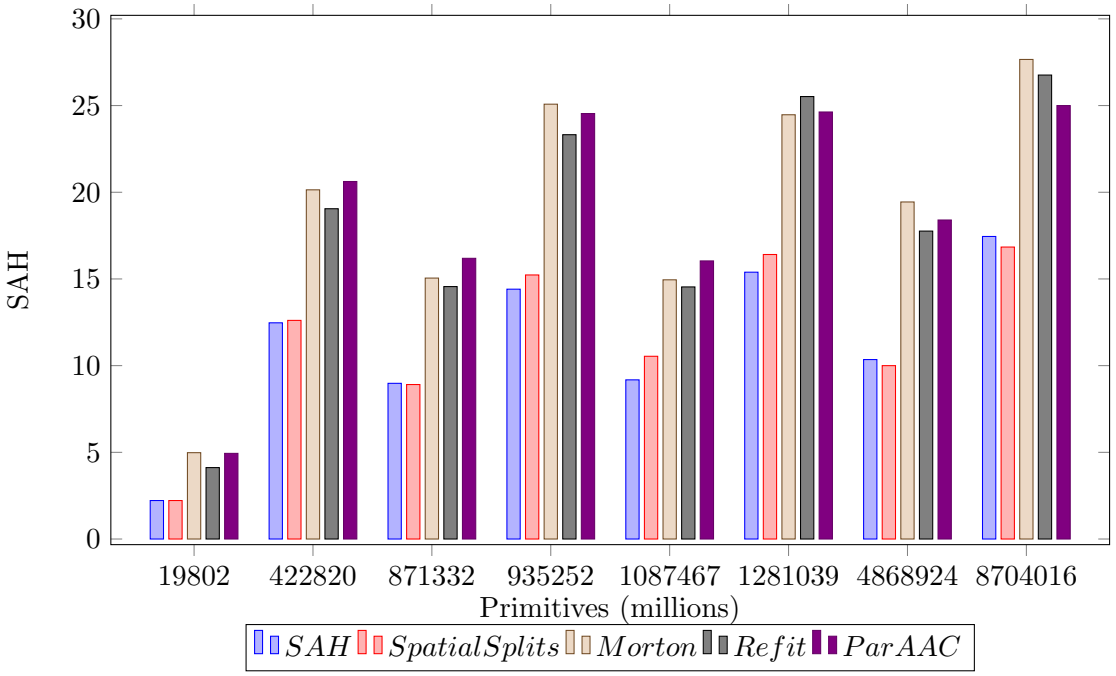


FIGURE 4.6: BVH Build Quality

Model	SAH	*SpatialSplits*	*Morton*	*Refit*	*ParAAC*
Gold Sphere	2.22	2.22	4.98	4.12	4.94
Dragon Bust	12.47	12.61	20.14	19.05	20.62
Dragon	8.98	8.91	15.05	14.56	16.19
Demon	14.41	15.23	25.08	23.32	24.54
Buddha	9.18	10.54	14.95	14.54	16.04
Woola	15.39	16.41	24.47	25.52	24.63
Imperial Crown of Austria	10.35	10.00	19.44	17.76	18.40
Dragon Bust B	17.45	16.84	27.66	26.76	25.00

TABLE 4.4: BVH Build Quality

after optimisation. The AAC algorithm also performed better than the Spatial Splits builder which is another high quality builder as the SAH. These three builders were below the pure Morton and Refit builders performance times which was expected as they are designed for high performance and speed. Our AAC approach could also benefit from some modifications to the implementation such as an alternative approach to storing clusters as vectors.

In terms of build quality the AAC algorithm's hierarchies are on par with or slightly below the Refit and Pure Morton builds with mesh primitive counts of one million and below. Once the model or scene's complexity exceeds the one million mark the AAC

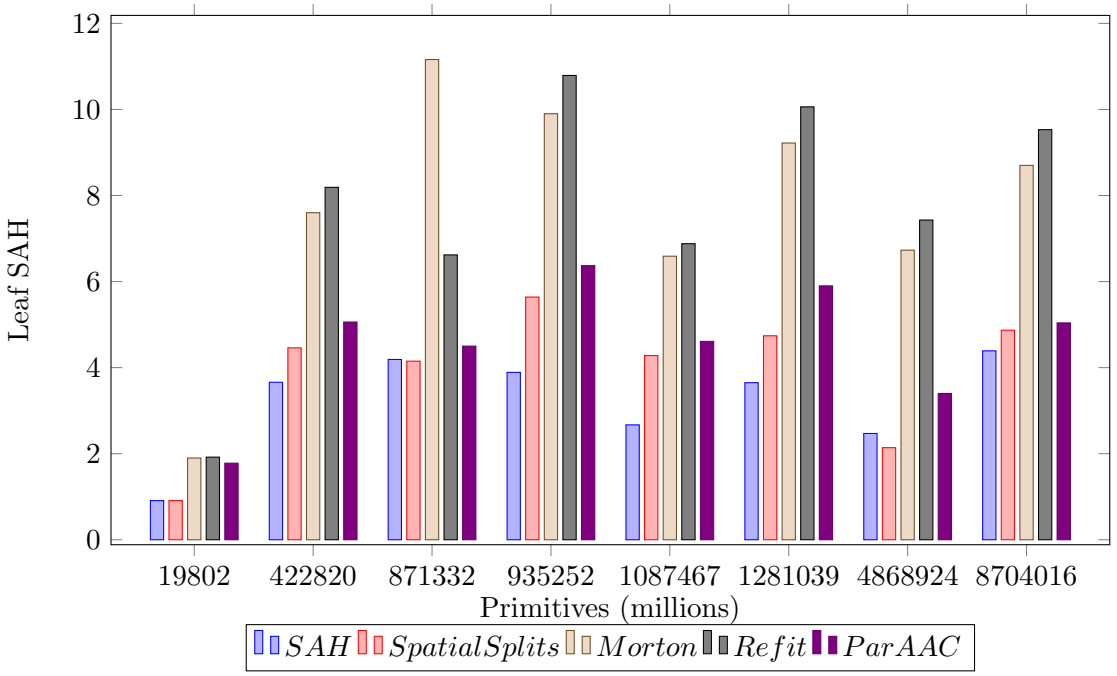


FIGURE 4.7: BVH Build Quality (Leaves).

Model	SAH	*SpatialSplits*	*Morton*	*Refit*	*ParAAC*
Gold Sphere	0.91	0.91	1.90	1.92	1.78
Dragon Bust	3.66	4.46	7.60	8.19	5.06
Dragon	4.19	4.15	11.16	6.62	4.50
Demon	3.89	5.64	9.90	10.79	6.37
Buddha	2.67	4.28	6.59	6.88	4.61
Woola	3.65	4.74	9.22	10.06	5.90
Imperial Crown of Austria	2.47	2.14	6.73	7.43	3.40
Dragon Bust B	4.39	4.87	8.70	9.53	5.04

TABLE 4.5: BVH Build Quality (Leaves)

algorithm's build quality also begins to rise and surpasses one or both of the high performance builders in the three most complex scenes that were tested. Though still a step below high quality builders like the binned SAH and Spatial Splits, this paper's simple implementation could benefit substantially in terms of build quality from some optimisation and tweaking of algorithm variables or the algorithm's nearest neighbour search procedure. As Gu et al. [11] report that the AAC-Fast implementation (thresholds organised for fast build) sacrifices a surprisingly small percentage of quality for increased performance, AAC was tested with thresholds which represent a fast build. Experimentation of these thresholds could potentially increase the quality of the hierarchies.

The SAH figures recorded for the leaves of the hierarchies are also positive for the AAC

algorithm. As our implementation creates leaves with single primitives and does not have a splitting threshold like the pure Morton builder for example, the leaf SAH values are surprisingly high while the overall SAH could be higher. The splitting threshold used in builders like the pure Morton builder means the splitting stops once this threshold is reached and leaves are then created with the primitives in this group, meaning there are less nodes created (lower tree depth), decreasing overall ray cost, but meaning the leaf intersection will be generally higher as they are not as well defined as the AAC implementation. Our implementations beats the leaf quality of the two high performance builders and is just short of the high quality builders leaf SAHs with all scenes and models.

Overall, a relatively unoptimised implementation of the AAC algorithm can be seen to produce BVH structures of a competitive quality at a surprisingly efficient rate compared to other more established algorithms. Our AAC implementation has shown consistency in both its performance and its build quality and with some optimisation, could be significantly faster than the high quality builders and produce hierarchies of similar quality.

Chapter 5

Conclusion

5.1 Future Work

In terms of future work, there are many avenues which could be taken in order to improve the Approximate Agglomerative Clustering algorithm within the Embree ray tracing kernels. The full utilisation of the kernel feature set could significantly increase construction performance as well as build quality.

Much of the algorithm's potential may also have been stifled by our own personal implementation. It would be very be interesting to see what performance improvements could be achieved by replacing the cluster vector approach with something more elegant and by thoroughly optimising the procedures of the algorithm. It would also be interesting to see the effects that tweaking the algorithm thresholds may have on both build performance and quality.

As well as making improvements to our own implementation, Gu et al. [11] suggest the investigation of a "lazy" variant of the AAC algorithm where it may be possible to leverage the top-down bisection process of the build to achieve laziness.

In terms of evaluation of the work, the AAC algorithm could be tested in an animated scene using refits and periodic full rebuilds once the build time of the refits has become greater than that of a full AAC rebuild. Unfortunately this scenario was not readily available to produce within Embree but is something that would be interesting to look at in future.

5.2 Conclusion

As Approximate Agglomerative Clustering is a relatively new and unstudied BVH construction algorithm, this research presents an implementation of the algorithm in the Embree ray tracing kernels in order to observe and evaluate the effectiveness and potential of the algorithm within the kernels and how it may utilise the kernel's features in order to contend with other, more established construction techniques. These aims were achieved by analysing some important statistical figures of the BVHs that the implementation produced such as the build times and the SAH of the produced hierarchies. These figures were compared against a number of different algorithms and a number of different models to achieve a well-rounded evaluation.

The findings in the evaluation of this implementation suggest AAC to be a promising algorithm which should continue to be researched and considered a significant competitor to other construction techniques.

Although BVH hierarchies of the same quality reported by Gu et al. [11] were not achieved, many avenues for improvement are available, through the Embree kernels and otherwise, and quality could be improved. As well as a positive outlook on BVH quality, the performance of the builder was also very competitive.

This paper has shown that a straightforward implementation of the Approximate Agglomerative Clustering algorithm can produce BVH structures of competitive quality and with efficient performance compared to more established techniques. The AAC implementation presented has demonstrated consistency in both its performance and its build quality and with some optimisation, could be significantly faster than some of the high quality builders and produce hierarchies of similar quality. As a result, this research has shown that Approximate Agglomerative Clusering and Approximate Clustering in general is a promising BVH construction technique with a lot of potential which the research community and practitioners should be aware of.

Bibliography

[1] J. Amanatides, A. Woo, *A Fast Voxel Traversal Algorithm for Ray Tracing*, In Eurographics '87, Eurographics Association, 1987, 3–10.

[2] C. Benthin, I. Wald, S. Woop, M. Ernst, W.R. Mark, *Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture*, IEEE Transactions on Visualization and Computer Graphics 18, 9, 2012.

[3] M.J. Doyle, C. Fowler and M. Manzke, *A Hardware Unit for Fast SAH-optimised BVH Construction*, ACM Transactions on Graphics, Vol. 32, No. 4, Article 139, 2013.

[4] C. Ericson, *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*, CRC Press, 2004, 236–237.

[5] M. Ernst, *Embree: Photo-Realistic Ray Tracing Kernels*, available at https://software.intel.com/en-us/articles/embree-photo-realistic-ray-tracing-kernels, 3 August 2012, last accessed 1 May 2014.

[6] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam and D. Burger, *Dark Silicon and the End of Multicore Scaling*, Proceedings of the 38th Annual International Symposium on Computer Architecture, 2011, 365–376.

[7] L. Feng, *Introdcution to Embree 2.1 - Part 1*, available at `https://software.intel.com/en-us/blogs/2014/01/24/introduction-to-embree-21-part-1`, 24 January 2014, last accessed 1 May 2014.

[8] K. Garanzha, J. Pantaleoni and D. McCallister, *Simpler and Faster HLBVH with Work Queues*, Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, 2011, 59–64.

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

[10] S.A. Green, D.J. Paddon, *A Highly Flexible Multiprocessor Solution for Ray Tracing*, The Visual Computer, 1990, 6(2):62–73.

[11] Y. Gu, Y. He, K. Fatahalian, G. Blelloch, *Efficient BVH Construction via Approximate Agglomerative Clustering*, High Performance Graphics, 2013.

[12] Intel Corporation, *Embree API*, available at `http://embree.github.io/api.html`, 2013, last accessed 3 May 2014.

[13] T. Ize, I. Wald, S.G. Parker, *Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures*, In Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualisation, 2007.

[14] T. Karras, *Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees*, High Performance Graphics, 2012, 33-37.

[15] T. Karras, *Thinking Parallel, Part III: Tree Construction on the GPU*, available at `http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/`, 19 December 2013, last accessed 13 May 2014.

[16] A. Kensler, P. Shirley, *Optimizing Ray-Triangle Intersection via Automated Search*, IEEE Symposium on Interactive Ray Tracing, 2006, 33–38.

[17] A. Lagae, P. Dutré, *Compact, Fast and Robust Grids for Ray Tracing*, Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Renderings) 27, 4, 2008, 1235–1244.

[18] C. Lauterbach, S. E. Yoon, D. Tuft and D. Manocha, *RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes Using BVHs*, IEEE Symposium on Interactive Ray Tracing, 2006., 39–46.

[19] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke and D. Manocha, *Fast BVH Construction on GPUs*, Eurographics 2009, Vol. 28, No. 2, 2009.

[20] D.J. MacDonald, K.S. Booth, *Heurisitcs for ray tracing using space subdivision*, The Visual Computer 6, 3, 1990, 153–166.

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

[22] J. Pantaleoni and D. Luebke, *HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry*, Proceedings of the Conference on High Performance Graphics, 2010, 87–95.

[23] S.G. Parker, W. Martin, P.P. Sloan, P. Shirley, B.E. Smits, C.D. Hansen, *Interactive ray tracing*, In Proceedings of Interactive 3D Graphics, 1999.

[24] M. Reshetov, A. Soupikov and J. Hurley, *Mulit-Level Ray Tracing Algorithm*, ACM Transaction on Graphics (Proceedings of ACM SIGGRAPH), 2005, 24(3):1176–1185.

[25] D. Sopin, D. Bogolepov and D. Ulyanov, *Real-time SAH BVH Construction for Ray Tracing Dynamic Scenes*, Proceedings of the 21st International Conference on Computer Graphics and Vision (GraphiCon), 2011.

[26] M. Stich, H. Friedrich, A. Dietrich, *Spatial splits in Bounding Volume Hierarchies*, In Proceedings of High Performance Graphics, 2009, 7–13.

[27] C. Wächter, A. Keller, *Instant Ray Tracing: The Bounding Interval Hierarchy*, In Rendering Techniques 2006 - Proceedings of the 17th Eurographics Symposium on Rendering, 2006, 139–149.

[28] I. Wald, V. Havran, *On Building Fast k-D Trees for Ray Tracing and On Doing That in O(NlogN)*, In Proceedings of Interactive Ray Tracing, 2006, 61–69.

[29] I. Wald, *Fast Construction of SAH-Based Bounding Volume Hierarchies*, Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing, 2007, 33–40.

[30] I. Wald, S. Boulos, P. Shirley, *Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies*, ACM Transactions on Graphics, 2007, 26(1):1–18.

[31] I. Wald, *Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architetcure*, IEEE Transactions on Visualization and Computer Graphics, 2012, 47–57.

[32] I. Wald, S. Woop, C. Benthin, G.S. Johnson, M. Ernst, *Embree: A Kernel Framework for Efficient CPU Ray Tracing*, ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH), 2014.

[33] B. Walter, K. Bala, M. Kulkarni, K. Pingali, *Fast Agglomerative Clustering for Rendering*, IEEE Symposium on Interactive Ray Tracing, 2008, 81–86.

[34] Wikipedia, *Bounding Volume Hierarchy*, available at `http://en.wikipedia.org/wiki/Bounding_volume_hierarchy`, last updated 21 June 2013.

[35] S. Woop, C. Benthin, I. Wald, *Embree Ray Tracing Kernels for the Intel Xeon and Intel Xeon Phi Architectures*, available at `http://embree.github.io/data/embree-siggraph-2013-final.pdf`, 2013, last accessed 2 May 2014.

[36] S.E. Yoon, S. Curtis, D. Manocha, *Ray Tracing Dynamic Scenes using Selective Restructuring*, In Eurographics Symposium on Rendering, 2007.