ANALYZING PERFORMANCE OF BOUNDING VOLUME HIERARCHIES FOR RAY TRACING

by

KEVIN W. BEICK

A THESIS

Presented to the Department of Computer and Information Science
and the Robert D. Clark Honors College
in partial fulfillment of the requirements for the degree of
Bachelor of Arts

December 2014

An Abstract of the Thesis of

Kevin Beick for the degree of Bachelor of Arts

in the Department of Computer and Information Science to be taken December 2014

Title: Analyzing Performance of Bounding Volume Hierarchies for Ray Tracing

Approved: 

Professor Hank Childs

A Bounding Volume Hierarchy — a type of informational data structure commonly used in computer graphics — is a popular means of accelerating the ray tracing algorithm used to render 3D images. There are many possible variations to consider when implementing a BVH for a particular ray tracing project. The goal of this thesis is to gain an understanding of how a few of the most significant design decisions affect a BVH's performance. To gain a thorough understanding of the internal mechanics of BVHs, I wrote and assembled my own codebase that records runtime metrics during the execution of the ray tracing algorithm.

Acknowledgements

I would like to thank Professor Childs for providing guidance and encouraging me to achieve my full potential as well as for being flexible and accommodating despite having many other duties throughout this process that started well over a year ago. I would also like to thank Matthew Larsen for providing me with some starter code, sharing his knowledge and directing me towards useful resources and Professor Hopkins for her willingness to serve as my CHC Representative on such short notice.

Table of Contents

List of Figures

List of Tables

Introduction

Computer Graphics

The fundamental purpose of a computer is to execute simple arithmetic and logical operations. The countless combinations of these operations make possible the seemingly infinite capabilities of the modern computer. A computer's processor is the piece of hardware responsible for executing the instructions of a program by performing various basic operations, and although the potential workload of modern processors is very large, it is limited to a finite number of operations per second.

The rendering of computer graphics — be it a desktop interface, a virtual reality or the visualization of scientific data — follows the same generic protocol as any other type of computer processing. Put simply, the computer does various calculations to determine the coloring of each pixel of the image to be produced and then refreshes the screen with that new image. Any "movement" that takes place on a screen is really a progression through a series of subtly different static images. This is true whether a window is dragged across a desktop interface or an object in a game is rotated; the computer is constantly refreshing the screen with new images to stay current with the user's actions.

The quicker a computer generates and displays new images, the smoother the resulting animation is. The metric normally associated with this concept is a device's frame rate or frames per second (FPS) — the rate at which new, consecutive images (frames) are produced. Increasing FPS is analogous to increasing the density of pages in a flip book; the first and last images remain unchanged, but as the number of intermediate pages increases, so does the fluidity of the animation.

When tasked with processing high quality graphics, the hardware's finite constraints become relevant. Rendering photorealistic graphics at high frame rates requires considerable computational power, often too much for a single computer. Therefore, it is typical to make qualitative sacrifices to ensure other performance benchmarks are met. One such potential benchmark is a real-time constraint. In order to be considered real-time, a program must guarantee a response or completion of a process within a strict time limit regardless of the workload [Ben90]. With real-time graphics, one can expect prompt visual reaction in response to any user action.

Rasterization

Rasterization is one of the most popular techniques for producing 3D graphics because it enables real-time rendering. However, rasterization does not allow for photorealistic lighting, resulting in images that, while sufficiently clear and accurate, are obviously computer generated. Nevertheless, its ability to quickly and reliably create visually pleasing images has helped to establish rasterization as the traditional method of rendering 3D graphics for the last several years.

Rasterization is a process that takes a three-dimensional scene and transforms it into a flat image. In computer graphics, the conventional means of representing a three-dimensional model is to store the geometry as a mesh of very small triangles, each with its three vertices in three-dimensional space. From any particular view (where the "camera" or virtual eye is situated in the scene and how it is oriented in space), each vertex of each triangle corresponds to a two-dimensional point in that view's image plane. To help illustrate this idea, imagine taking a portrait photo of a person posing in front of a distant mountain. When that photo is printed to paper, every visible point in

the captured scene, no matter its distance from the camera, maps to a two-dimensional point on the paper. Rasterization follows this basic concept, mapping every triangle's vertices to two-dimensional points dependent on the current view while monitoring the layering of the geometry, essentially flattening the scene. After processing the entire geometry of the scene, the final step of the rasterization process is to determine how the visible triangles of the flattened scene sit on the image plane and overlap its pixels. For any pixel that does indeed include a triangle, the appropriate color — including simple lighting — is calculated and added to the image buffer. Finally, after calculating each pixel's color, the image file is written.

Because rasterization processes the entirety of a scene's geometry, the quantified workload is heavily associated with the complexity of the whole scene, that is, the number of triangles that constitute the scene.

Ray Tracing

Ray Tracing is another technique used in rendering 3D graphics. What sets ray tracing apart from rasterization and other techniques is the high degree of realism for which it allows, enabled by precisely accounting for many real-world lighting effects. This higher quality is naturally accompanied by an inherent greater computational cost. The process cannot guarantee results within a strict timeframe and thus is not a real-time method. Nevertheless, its realistic results make it an excellent choice when render time is not a concern; the productions of still images and animated movies and television shows typically make use of ray tracing in their rendering processes.

The greater realism of ray tracing comes from its ability to accurately simulate real-world lighting effects such as reflections, shadows and global illumination. As

demonstrated in Figure 1, the technique involves tracing the path of a light ray from the "camera" through each pixel of the image plane and then calculating the color of each pixel based on the various interactions of the respective light rays with the scenery. This process is known as intersection detection. Whereas rasterization processes the entire scene and converts it to a two-dimensional representation, ray tracing works on each pixel separately and successively, determining which elements of the scene are depicted in each. Hence, only the portion of the scene that appears in the image — based on the camera position and orientation — is processed.

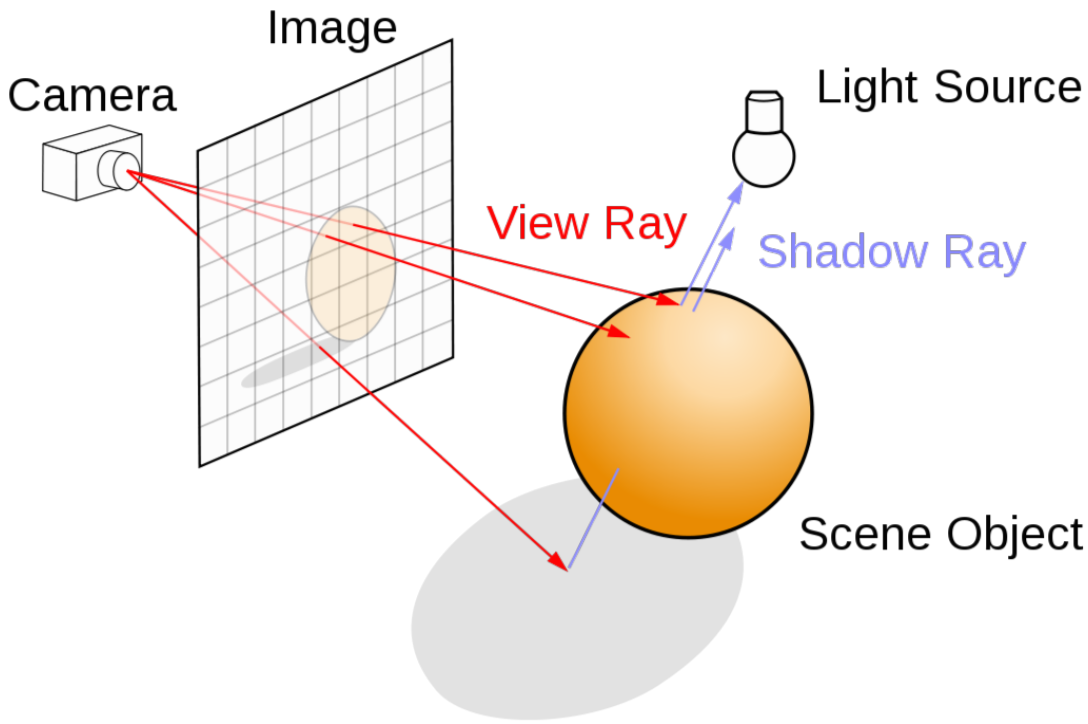

Figure 1: A simple illustration of ray tracing's intersection detection.

[Adapted from the Wikimedia Commons file "File:Ray trace diagram.svg" http://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg]

If a ray intersects an object, then additional rays can be cast from the point of intersection to assess the surrounding space; shadows, reflections, refractions, global illumination and any other lighting effects are determined by these secondary rays,

allowing for precise adjustments to the absolute color of the intersected object in each pixel. To enhance quality or realism, the traces can go "deeper" to better approximate real-world lighting. Of course, this also increases the complexity and number of necessary calculations. The workload to generate an image using ray tracing is largely determined by the resolution of the desired image (how many pixels are there?) and the depth of the traces during the intersection detection phase (how many rays are cast for each pixel?).

The notion that ray tracing could be a viable rendering technique in highly responsive software, such as that used for scientific visualization, has long been dismissed because of ray tracing's relatively slow performance. However, due to continuing scientific and technological advances, it is not only possible to generate immense data sets (on the order of billions of data points), but often necessary for certain disciplines and research. Consequently, the geometries that represent such data sets are larger and more complex which, in turn, makes ray tracing increasingly cost effective. As previously noted, while the costs associated with rasterization are proportional to the geometry's size, ray tracing's costs are proportional to the image size (i.e., the number of pixels) [Nav13]. Thus, research in graphics performance has gradually been shifting its focus towards improving the ray tracing algorithm.

Various acceleration techniques and data structures have been devised to minimize the time associated with ray tracing while maintaining the aspects that produce high-quality results. The Bounding Volume Hierarchy (BVH) is a versatile and highly effective acceleration data structure, making it one of the most interesting acceleration techniques. By organizing the geometry of scene in a spatial hierarchy,

BVHs lead to drastic cuts in time spent during the longest phase of ray tracing, intersection detection.

Bounding Volume Hierarchies

The Bounding Volume Hierarchy is a type of tree data structure that extends the ray tracing algorithm to improve the speed of intersection detection. Like all tree data structures, the BVH is a hierarchical organization of nodes with its root node at the top and the root's child nodes directly beneath it. Any child nodes may have children of their own, increasing the height of the tree with each level of nodes. It follows that each node aside from the root has exactly one parent node. At the bottom of the structure are the leaf nodes — those that do not have any children. Non-leaf nodes are known as internal nodes. The number of children directly under an internal node is referred to as that subtree's degree and is typically predetermined by the tree's designer. The degrees of the various subtrees throughout a given tree may or may not be uniform depending on the tree's implementation.

The tree structure of a BVH spatially organizes the primitive shapes of a geometry's meshes (e.g., small triangles) in an effort eliminate unnecessary work during ray tracing's intersection detection [Gun07]. The leaf nodes each represent a small set of proximate primitives and their minimum joint bounding volume. Nearby volumes are grouped and enclosed in larger bounding volumes which are represented in internal nodes. This pattern is consistent throughout the tree such that a single bounding volume encompassing the entire geometry is represented in the root node. Ray tracing makes use of the BVH during intersection detection by starting at the root node and traversing the descendent nodes that have their respective bounding volumes intersected by the

current ray and ignoring those in which no intersection occurs. This implicitly eliminates groups of triangles from consideration as intersection candidates as the traversal progresses [Gun07]. The main advantage in using a tree to organize the bounding volumes is the reduction of the intersection search's time complexity to a logarithm of the number of total primitives (the logarithm with base equal to the average tree degree).

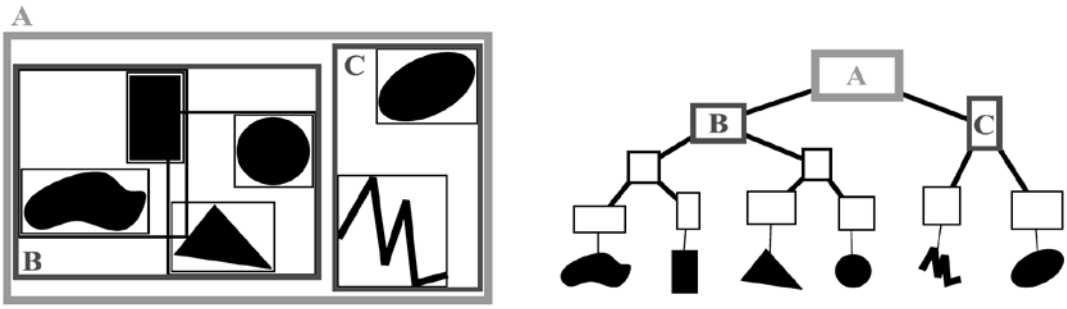

Figure 2: An example of a bounding volume hierarchy, using rectangles as bounding volumes.

Borrowed from "Results on Geometric Networks and Data Structures" pg. 10 [Hav04].

There are a number of trade-offs to consider when designing a BVH for a particular project. The shape of the bounding volumes is significant; simple shapes are easy to store and intersect but opting for a more complicated or flexible shape may allow for tighter bounding volumes (and less empty space in the volumes). The degree of the tree — also known as a BVH's branching factor — profoundly affects the traversal process. Fewer children per node leads to less work in determining which, if any, child to traverse, but also produces a taller tree with more nodes between the root and leaves. Conversely, a higher degree results in shallower leaf nodes and fewer nodes to traverse but more work at each node to determine which child to follow. Beyond these design decisions, there are many other properties to consider [Eric04]:

- The nodes contained in any given sub-tree should be near each other. The lower down the tree, the nearer the nodes should be to each other.
- Each node in the BVH should be of minimal volume.
- "Pruning" a node near the root of the tree removes more objects from further consideration.
- The volume of overlap of sibling nodes should be minimal.
- The BVH should be balanced with respect to both its node structure and its content. Balancing allows as much of the BVH as possible to be pruned whenever a branch is not traversed.

There are two standard approaches to constructing a BVH: the top-down and bottom-up paradigms. The top-down approach starts with an all-encompassing volume for the entire geometry in the root node. The volume is split into smaller, roughly equal, volumes; one for each of the root's children. This is repeated for each internal node until volumes small enough for leaf nodes are generated; the typical halting criterion is enclosing fewer than some maximum number of primitives per leaf. Alternatively, the bottom-up approach starts with leaf nodes and iteratively gathers them into larger volumes, continuing until a single root node has been produced. The top-down method is easy to implement and generally builds faster than bottom-up, however the latter usually produces better trees.

Related Works

BVHs have developed and evolved from the principles of other data structures over the past few decades. The k-dimensional tree (kd-tree), introduced by Bentley in 1975, is a partitioning binary tree for k-dimensional space. It "facilitates many different and seemingly unrelated query types," including intersection queries [Bnt75]. For this reason, the kd-tree and its derivatives have been a prevailing standard in ray tracing acceleration implementations. The R-tree, introduced by Guttman in 1984, is a data structure used for organizing geometric objects and laid some of the groundwork for the development of BVHs. This data structure groups spatially proximate objects and represents them with a minimum bounding rectangle. The trees are parameterized by the maximum degree of the nodes, chosen such that a node fills a full block on the computer's disk [Gut84].

Perhaps the most well-established heuristic used in BVHs is the axis-aligned bounding box (AABB). Their simple shape and ease of calculating intersections make AABB a popular choice for bounding volumes. Using AABBs, efficient construction algorithms have been developed that are "more robust than [other] heuristic approaches" [Hav04]. Another common heuristic is triangle splitting: the partitioning of geometric primitives (e.g., small triangles) in order to increase performance by eliminating unnecessary work. It has been demonstrated (by several studies) that splitting triangles before construction of the BVH structure itself can lead to faster processing across many different data sets [Ern07]. This improvement partially comes from resolving the issue of insufficiently "tight" bounding volumes by subdividing select triangles, allowing for bounding volumes that have less empty space. "A speedup

of up to a factor of 10 [can] be achieved while the number of triangle references increase[s] only by 16 percent" [Dam08].

Another popular heuristic is the Surface Area Heuristic (SAH). It assumes that the total surface area of objects is directly related to the likelihood of them being intersected by an arbitrary ray. An early construction algorithm for BVHs was able to reduce the number of nodes, leaves and objects visited by a ray by using SAH in conjunction with another heuristic for estimating the optimal splitting plane (between the spatial median and object median) [Mac90]. Another optimization algorithm for BVHs built using SAH used subtree rotations to reduce a tree's total cost [Ken08].

The most successful algorithms use some combination of techniques, taking advantage of the benefits of each. Cadet and Lecussan combined BVHs with binary space partitioning trees to take advantage of the best properties of each, resulting in a 50% increase in processing speed over the unaltered structure [Cad07]. Shuai et al. developed a BVH construction algorithm that builds using various strategies: shallow levels use spatial partitioning while deep levels utilize object partitioning. Their results indicate that this new algorithm takes less than 40 percent of the build time as compared to similar quality BVHs using the latest spatial splitting BVH algorithm [Shu13]. Wei et al. combined the advantages of boxes and spheres as bounding volumes, along with using parallel programming model for traversal. This algorithm outperformed other collision detection algorithms in efficiency and accuracy, thus meeting "the real-time and [accuracy] requirements in complex interactive virtual environment" [Wei08]. Another algorithm uses BVH and a "feature-based method," implementing a

combination of spheres, AABB, and custom volumes as bounding volumes, which promotes tighter bounding as objects deform and move [Mad09].

As real-time ray tracing has become more practical in the last ten years or so, the issue of deformed or animated models has started receiving more attention. An algorithm developed in 2006, specifically for such models, takes advantage of a new (at the time) method to help detect when a BVH needs to be rebuilt. This algorithm was able to achieve up to 13 frames per second, while including secondary rays [Lau06]. This study also demonstrated that ray coherence techniques introduced for kd-trees can be extended to BVHs and yield similar improvements [Lau06]. As alluded to earlier, kd-trees have long been the standard-bearer for fast rendering of static models. However, the gap is quickly closing — or perhaps it has already closed — between kd-trees and BVHs in terms of performance on static data sets, thanks in part to the application of kd-tree improvements to BVHs. For example, Wald argues that fast approximation construction techniques, originally proposed for kd-trees, actually provide a greater performance boost when applied to BVHs [Wal07].

One of the keys to achieving high performance in ray tracing is the idea of "Single Instruction, Multiple Data" (SIMD), that is, performing the same operation on multiple data at one time. Ernst and Greiner, inspired by this principle, developed a new data structure they called a Multi-BVH. An MBVH collapses BVH subtrees of height two into SIMD nodes that store four bounding boxes. This allows for data-level parallelism during traversals and triangle intersections, yielding speedups of up to 2.8 times while using less memory than regular BVHs and without modification to the architecture of the rendering engine [Ern08]. Wald et al. took their study in the

opposite direction when they examined traversals of individual rays through BVHs that had sixteen children per node — the typical BVH traversal procedure involves intersecting packets of coherent rays. They found that while this was less efficient for primary rays, it turned out to be comparable to, if not faster than, typical packet traversal techniques for less "coherent secondary ray distributions" [Wal08].

Wachter and Keller presented a new termination criterion for the common top-down construction method of BVHs in 2007. It allows for *a priori* fixing of the data structure's memory footprint, which results in a shorter, more efficient algorithm that automatically balances during construction [Wac07]. In 2013, Wu et al. reduced unnecessary ray computations in subspaces, leading to faster performance [Wu13]. Another big advancement came with a new, special BVH structure that is particularly valuable for "massive" models. The tree is compressed and can selectively decompress nodes without decompressing the entire BVH. This approach results in performance that is four times faster compared to using uncompressed data [Tae10].

While kd-trees have a reputation of being efficient with static models, BVHs have always been the best option when dealing with dynamic data because of how simple it is to update an existing BVH. Since BVHs are the common choice for dynamic scene, studying the potential of updating preexisting BVHs is prominent and essential. Reorganizing subtrees to optimize performance is an example of "modifying an existing BVH to improve its quality" [Kar13]. Another example is Garanzha's algorithm from 2008 that efficiently updates BVHs, developed to be effective with highly dynamic scenes. This algorithm tries to focus on less costly operations by working with higher nodes, which encompass triangles that do not move as much

relative to each other. His algorithm has an update time that is two to four times faster than that of other popular techniques [Gar08].

Graphics rendering is expensive and specialized, and thus most of the computation is done on the GPU. However, studies have been carried out that explore BVH performance on CPUs. A “fast CPU-based” BVH construction algorithm has been developed that approximates the SAH and whose performance approaches that of kd-trees [Gun07]. Also, an efficient method on the CPU that uses coherent packet traversal of an optimally “pruned” BVH, while not as fast as GPU methods, outperforms other “out-of-core-GPU methods by orders of magnitude” [Kno11]. Furthermore, modern ray tracing frameworks, such as Embree, that are designed to “maximize utilization of modern CPU architectures” perform comparably with existing GPU methods [Wal14].

Experiment Overview

It is clear that including a BVH in a ray tracing implementation increases the speed of intersection detection, but not all BVHs are the same. BVHs with differing structures and internal organization surely have different operational tendencies. The goal of this thesis is to gain an understanding of how various BVH design decisions affect performance during ray tracing. As previously stated, a BVH is just one piece of an image rendering program, so in order to work within a self-contained project, as well as for my own edification and experience, I write my own codebase including data loader/processor, various BVH constructors, ray tracing engine, image file generator, and assorted auxiliary functions.

BVH Variants

There are many possible BVH variants, some of which have been considered in the earlier sections of this thesis. For the purposes of this study, I focus on two significant design decisions: the degree of the tree and the construction technique. More precisely, my code supports BVHs that have a branching factor of two, four or eight, plus BVH construction by either the top-down or bottom-up method. These are two of the most common and meaningful variations and thus are the focus of my investigation.

Another significant design decision is the shape of the bounding volumes. There is a myriad of possible bounding shapes and thoroughly studying this factor would require its own investigation. Therefore, my BVHs use exclusively AABBs. As mentioned above, this is a well-established heuristic due to the simple nature of the AABB. Although bounding volume shape is not considered in this study, it is certainly

a significant BVH characteristic and its affects on performance could be incorporated into future work.

Test Cases

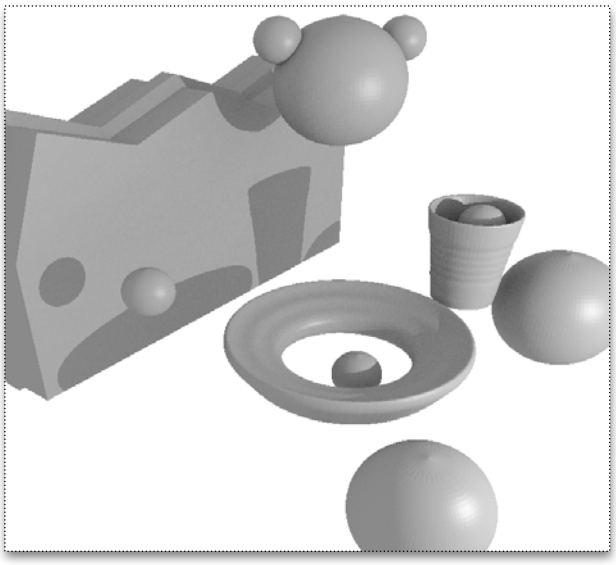

Figure 3: A rendering of one of the five input geometries processed during the ray tracing tests.

All tests involve one of five different versions of this scene, each scene only differing in the number of triangles used to represent the geometry. Pictured is the largest version, made up of 88,748 triangles.

To analyze the relative performances of the six BVHs (two construction techniques with three branching factors), each is used in ray tracing trials to assist in the rendering of five different geometries. The five geometries represent the same scene, but the complexity of each (the number of triangles that compose each model) differs.

Model Name	Tiny	Small	Medium	Large	Huge
Number of Triangles	648	2,540	11,516	25,340	88,748

Table 1: Geometry sizes.

There are five geometric representations of the scene depicted in Figure 3, each composed of a different number of triangles (complexity).

To gain further insight, I run the ray tracing algorithm with three different lighting effects:

1. A minimum trace depth with full opacity. That is to say, no additional rays are cast beyond those which originate at the view point.
2. A minimum trace depth with half opacity.
3. A trace depth of two (single reflection ray) with full opacity.

In all, this results in 90 test cases: each combination of the five geometries, three branching factors, two construction techniques, and three lighting effects.

Analyzing Performance

I run my tests on my 2009 iMac, which has a 2.93 GHz Intel Core 2 Duo processor. It is typical to carry out graphics computations on GPUs as these are designed specifically to handle such workloads. However, state-of-the-art algorithms and frameworks, such as Embree, are leading to an increase in CPU-based ray tracing [Wal14]. I follow this trend in my investigation, running my tests on my iMac's CPU. My CPU-based implementations are not necessarily competitive with modern GPU methods, yet their performance metrics should prove insightful in my study. I will not be concerned with my ray tracer's absolute performance benchmarks, but rather the *relative* performances of the BVHs with respect to each other. I assume the various trees' relative performances are consistent independent of whether they are built and utilized on my CPU or a different piece of hardware. Thus the goal of this thesis — to better understand how various BVH design decisions affect performance — is indeed achievable using my CPU.

I record three metrics to analyze performance, the first of which is the BVHs' build times. Each of the six BVH categories I work with are structurally different and

thus require different amounts of time to complete construction. The relative differences in build times, as opposed to the build times themselves, are indicative of the relative computational costs to construct the various BVHs. The other two metrics measure the work involved in traversing a tree and are representative of a tree's "quality." These metrics are the average time spent generating each pixel's color and the average number of BVH nodes visited per pixel.

Construction Algorithms

To split a node into two child nodes, my top-down construction method divides a node's geometry with a plane that is perpendicular to either the x-, y- or z-axis. It calculates the surface areas of potential new bounding volumes in an effort to minimize the empty space that ends up being encapsulated within a node. This algorithm iterates through 100 planes per dimension and selects the splitting-plane that minimizes the total surface area of the new children's volumes. It is not necessary to find the absolute best splitting-plane; doing so would take an unreasonable amount of time. For top-down construction, selecting the best plane of an evenly-spaced set of 100 sufficiently balances time consumption and tree quality. The nodes are split until there are at most four triangles per leaf node; this four triangle maximum in leaf nodes is consistent for all BVHs in this investigation.

For BVHs of branching factor four or eight, the top-down construction follows similar logic. The biggest difference is that instead of one splitting-plane, there are three when the branching factor is four and seven when the branching factor is eight. The objective of splitting a node into multiple children by finding the optimal divisions remains constant, but doing so with multiple splitting-planes is substantially more

complicated and time consuming because of the much greater number of potential split combinations. For this reason, the number of potential splitting-planes is reduced proportionally to the branching factor: for branching factors of four and eight, the best combination of three planes from 50 candidate planes per dimension and of seven from 25 candidate planes per dimension, respectively, are selected to split an interior node. The selection process for a branching factor of eight is analogous to selecting the best combination of seven ingredients from 75 potential ingredients by iterating through and testing each possible combination. It is a long but necessary process to ensure the end result is a tree of sufficient quality.

My bottom-up implementations are inspired by the algorithm for binary BVHs proposed by Walter et al. [Wtr08]. Bottom-up construction starts the same way regardless of the branching factor, enclosing each individual triangle in its own node. These nodes are inserted into a kd-tree to afford quick access to proximate nodes. As Walter et al. discuss, "It may seem counter-intuitive that we use one hierarchical clustering tree of the data the build another. The idea is to use a simple-to-build, but lower quality, clustering tree to bootstrap the construction of a higher quality tree" [Wtr08]. The two most proximate nodes are removed from the kd-tree and, depending on their collective number of enclosed triangles, have their content and data merged into a new leaf node (if together the nodes have four or fewer triangles) or become the children of a new parent node (otherwise). The new node is then added to the kd-tree and the next most proximate pair is removed. This continues until there is only one node left in the kd-tree. At this point, that lone node is in fact the root of the entire BVH and the construction of the binary tree is complete. To get a BVH of branching

factor four or eight, this binary tree is collapsed, bottom-up. The end result is a tree that does not necessarily have a uniform degree throughout. The collapsed trees have nodes of degrees between two and the branching factor. Because the degrees are not uniform, I record the average degree of the bottom-up tree's nodes to more precisely correlate the number of children per node with the traversal metrics (see table 3).

Results and Analysis

Build times are the obvious metric with which to start the analysis. The 30 constructions (five geometries, three branching factors, two construction methods) resulted in a range of build times covering several orders of magnitude. I capped build times at eleven days and unfortunately four of the constructions on the "huge" data set could not complete within the given time limit. These four cases included each of the three bottom-up constructions as well as the top-down, branching factor of eight construction. Although four BVHs were not completed, quantifying those results still helps to establish an understanding of the respective design decisions' effects on performance. That is to say, when considered alongside the data of the other BVHs, the fact that these BVHs required greater than eleven days to be built does indeed contribute to our understanding of the relative performances.

The recorded build times paint a clear picture (see Figure 4 and Table 2). The top-down binary BVH had by far the quickest construction. Not surprisingly, build times for both construction techniques increased as the models increased in size. For each doubling of the branching factor, the top-down construction took dramatically longer, increasing the build times by at least two orders of magnitude.

The productions of binary bottom-up BVHs took significantly longer than those of their top-down counterparts, and increasingly so as the models got larger. For example, while binary top-down construction of the "huge" BVH completed in about 90 seconds, the bottom-up version took in excess of eleven days. However, increasing the branching factors of the bottom-up BVHs required less than a five percent increase in total build time. Evidently, the tree collapsing routine used to increase a tree's degree

does not require a significant amount of time relative to the build times of the bottom-up binary trees. Hence, while top-down build times grow exponentially as the branching factor increases, bottom-up build times are largely uniform per model. When constructing trees of a branching factor of eight, the build times are actually less than those of the top-down method.

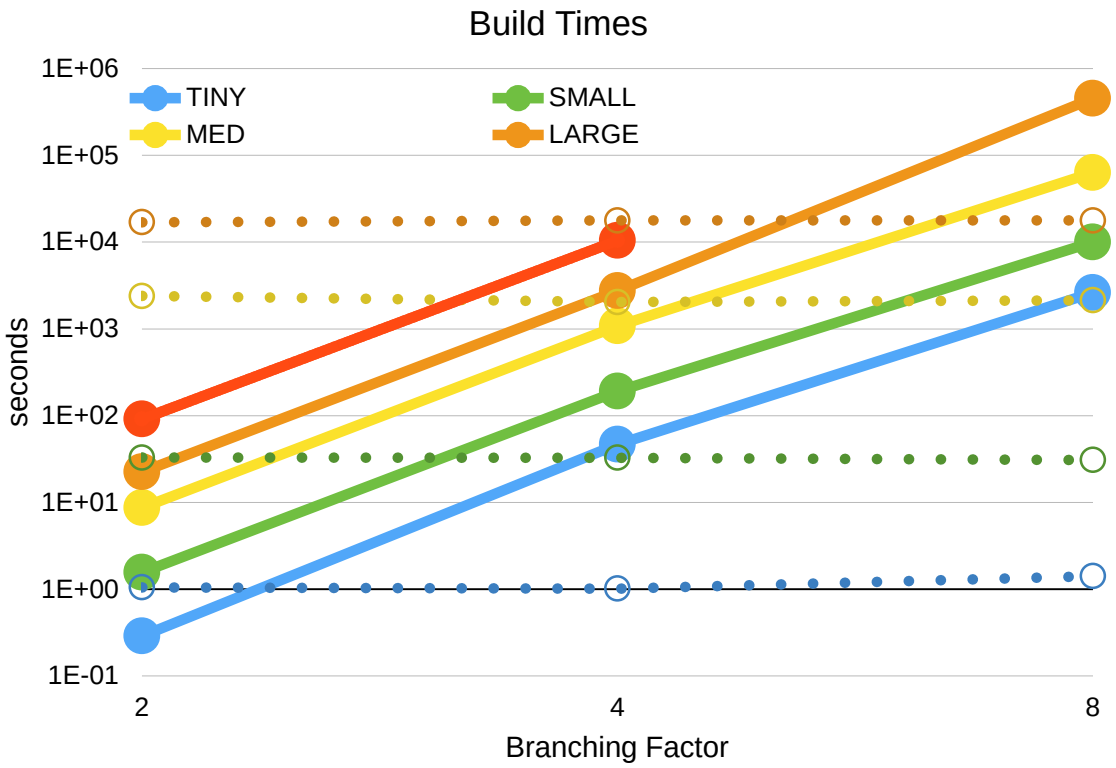

Figure 4: A graph of BVH build times.

The solid lines represent build time growths for the top-down approach while the dotted lines represent bottom-up build time growths. The five colors correspond to the five geometry sizes. Note: The "HUGE" point above branching factor 8 is a projected plot point based on the trend of the other "HUGE" points.

Top-down construction is the faster approach when building trees with branching factors of two or four. Bottom-up constructions for these two types of trees are remarkably slower, but they become more efficient as the degree of the desired BVH increases because the collapsing process is so quick. In fact, as the desired

branching factor increases from four to eight, bottom-up construction becomes the faster technique.

While build times are certainly important to understanding the performance of BVHs, the traversal metrics — traversal time per pixel and nodes visited per pixel — provide much more practical insight. These metrics determine the "quality" of the tree and its ability to meaningfully augment ray tracing's intersection detection. As anticipated, increasing the trace depth by incorporating reflections or transparency lengthens intersection detection, albeit slightly and appropriately so (see Tables 4-9). Across the board, neither the average time per pixel nor nodes traversed per pixel substantially increased when secondary rays were incorporated — each by only about 10-20%. Because each of my input models represents the same scene and the computation of lighting effects is greatly dependent on a scene's objects and its layout, one cannot discern whether this statistic is a faithful representative of what would be observed in different scenes or if it is particular to my test scene. Nevertheless, it is clear that given a relatively simple scene, secondary rays only moderately increase the computational workload. Since the relative performances of the BVHs are mirrored with each of the lighting effects, the following discussion does not explicitly differentiate the results of the three lighting effects.

Traversal time per pixel is indicative of the overall efficiency and organizational quality of a BVH. It is representative of the complexity of calculating which child node to pursue as well as the number of nodes for which it must do this calculation. In my tests, this metric demonstrated relative consistency in the smaller models while a more definite upward trend developed in the larger models (see Table 4). This is true for

trees from both construction techniques. This indicates that together the complexity and number of traversal calculations remained mostly balanced as the branching factor increased in the smaller models. In the larger models, this equilibrium was less balanced and the average traversal became less efficient as the branching factor increased.

The number of visited nodes per pixel describes traversal performance without assessing the calculations therein. Among top-down BVHs, those with a branching factor of four had the least number of nodes visited during intersection detection while those with a branching factor of eight had slightly more. The binary trees had the most nodes visited, by far. Meanwhile, bottom-up BVHs demonstrated a steady, consistent decline in the number of nodes visited as the branching factor increased (see Figure 5 and Table 5).

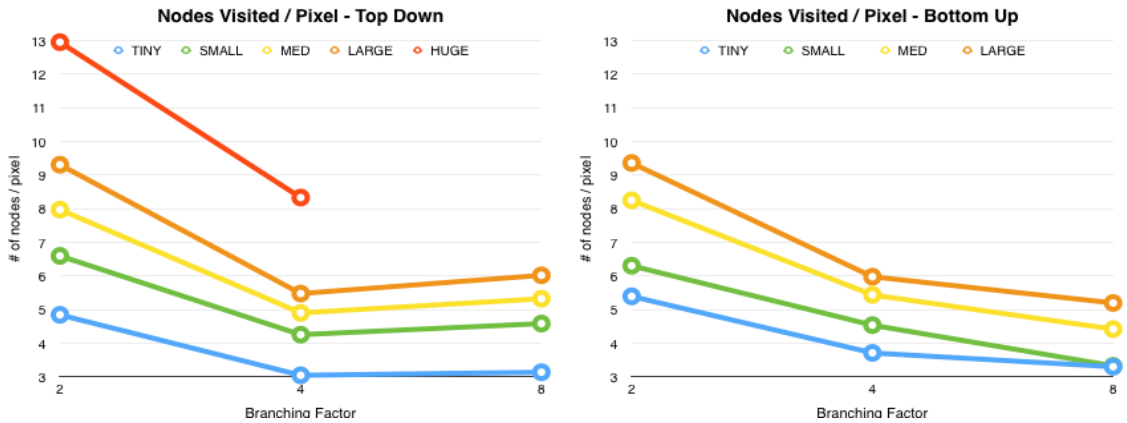

Figure 5: Graphs of the average number of nodes visited per pixel.

It is intuitive that increasing the average degree of a BVH would lead to fewer traversed nodes; a wider yet shallower tree has fewer nodes between its root and its leaves. Ultimately, this decrease in the number of visited nodes means fewer traversal calculations in which the proper child node to pursue is determined. On the other hand, increasing the number of children per node also increases the complexity of these

traversal calculations. When the two traversal metrics are considered together, it is apparent that the increase in net traversal times indicates that the extra computation involved in calculating which child node to pursue ultimately negates any potential efficiencies that might result from reducing the number of such calculations when using a shallower, wider tree.

Summary and Conclusion

This thesis explored the effects that a few significant design decision have on a Bounding Volume Hierarchy's performance. The investigation relied on a codebase I assembled, mostly composed of original code written by me. The BVH performance metrics came from data recordings captured before and during ray tracing renderings of five geometries of varying complexity.

Because my tests were run on my computer's CPU (as opposed to hardware specialized in rendering graphics), I exclusively analyzed the performances of the various BVH configurations relative to each other. This approach afforded me reliable and insightful results.

The top-down construction algorithm worked quickly and well for binary BVHs, but when extended to allow for trees of a greater branching factor the workload was such that sacrifices to the quality of the tree were necessary to allow for reasonable build times. Yet even with this handicap, the trees with higher branching factors yielded improved performance. If these top-down BVHs were built without a sacrificial reduction in quality (or at least to a lesser extent), I am confident that a top-down tree of a higher branching factor would significantly out-perform its counterpart beyond the performance gaps demonstrated by my BVHs. Of course a higher branching factor would still necessitate a longer build time, but if built once and then saved for multiple renderings it is reasonable to assume, based on the results herein, a tree of a higher branching factor would be worth the initial time investment.

The binary bottom-up constructor takes considerably longer than its top-down counterpart. Nevertheless, the trees generated by this implementation are solid and very

usable, as demonstrated by their performance metrics and the images they help to render. In fact, the bottom-up trees' performances were comparable to those of the top-down versions.

Perhaps the most compelling insight to take away from this investigation is that while collapsing a BVH to increase its branching factor does indeed result in fewer nodes traversed during intersection detection, it does not significantly increase the tree's total build time. However, this does not translate into a reduction in traversal times as increasing the degree comes with the cost of testing more child nodes for intersection per inner node. Hence, for processing that involves auxiliary operations for each traversed node, it may be worthwhile to collapse the BVH after its construction to minimize the number of nodes traversed.

Data Tables

* Build times were capped at eleven days. BVHs that could not be completed in the allotted time do not have associated traversal metrics.

Build Times (s)

Ti	640	0.20	46.87	2,613	0.64	0.65	0.66
Small	2,540	1.56	190.8	10,009	22.88	22.75	21.90
Med	11,516	8.72	1,082	62,216	2,040	2,041	2,128
Large	35,240	22.42	2,755	452,443	16,046	17,787	17,776
Huge	88,748	91.44	10,402	960,000+ *	960,000+ *	960,000+ *	960,000+ *

Table 2: BVH build times.

Unsurprisingly, build times increase dramatically as the model size increases. This is especially true in top-down BVHs with a higher branching factor. The collapsing technique used to generate bottom-up trees of a greater branching factor from binary trees does not significantly contribute to the build times.

Average Degree in Bottom-Up BVHs Created via

Ti	3.69	6.69
Small	3.63	6.63
Med	3.69	6.77
Large	3.71	6.71

Table 3: Average degrees of bottom-up BVHs created via collapsing.

The tree collapsing algorithm increases the average degree of a BVH's subtrees, resulting in a shallower tree. The maximum number of children each node can have is determined by the branching factor, although this number is rarely exactly met. This table displays the average degree of each parent node in the BVHs generated from the bottom-up collapsing algorithm. All BVHs not represented in this table have uniform degrees exactly equal to the corresponding branching factor.

Primary Rays - Ray Trace Depth: 1, Opacity: 0.0

Traversal Time per Pixel (µs) - Basic Lighting

Model		Top-Down (branching factor)			Bottom-Up (branching factor)		
Name	**Size**	**2**	**4**	**8**	**2**	**4**	**8**
Tiny	**648**	9.20	8.61	12.16	10.79	10.81	13.00
Small	**2,540**	20.08	17.43	21.54	22.26	22.67	24.25
Med	**11,516**	54.57	48.86	62.25	72.89	73.76	78.71
Large	**25,340**	140.93	155.47	207.92	131.14	141.97	157.59
Huge	**88,748**	881.86	1,200.86	*	*	*	*

Table 4: Average BVH traversal times for primary rays only.

The trend appears to be that increasing the branching factor also increases the average traversal time. This trend is clearer with the larger models.

Nodes Traversed per Pixel - Basic Lighting

Model		Top-Down (branching factor)			Bottom-Up (branching factor)		
Name	**Size**	**2**	**4**	**8**	**2**	**4**	**8**
Tiny	**648**	4.85	3.05	3.14	5.39	3.71	3.30
Small	**2,540**	6.60	4.26	4.59	6.30	4.54	3.33
Med	**11,516**	7.97	4.90	5.32	8.25	5.44	4.42
Large	**25,340**	9.31	5.48	6.02	9.36	5.98	5.20
Huge	**88,748**	12.95	8.33	*	*	*	*

Table 5: Average number of nodes traversed during intersection detection for primary rays only.

Of the top-down BVHs, those with a branching factor of 4 have the least number of nodes traversed during intersection detection while those with a branching factor 8 visit slightly more. However, in bottom-up BVHs, the number of nodes traversed demonstrates a steady, consistent downward trend as the branching factor increases.

Secondary Rays - Ray Trace Depth: 2, Opacity: 0.0

Traversal Time per Pixel (µs) - Trace Depth: 2

Model		Top-Down (branching factor)			Bottom-Up (branching factor)		
Name	**Size**	**2**	**4**	**8**	**2**	**4**	**8**
Tiny	**648**	12.65	9.35	12.41	13.52	14.08	17.16
Small	**2,540**	25.62	24.16	24.14	23.54	25.21	27.49
Med	**11,516**	61.46	53.79	63.53	90.91	83.29	90.49
Large	**25,340**	102.30	112.65	148.21	152.58	158.97	166.52
Huge	**88,748**	1,233.42	1,474.53	*	*	*	*

Table 6: Average BVH traversal times for primary plus reflection rays.

Nodes Traversed per Pixel - Trace Depth: 2

Model		Top-Down (branching factor)			Bottom-Up (branching factor)		
Name	**Size**	**2**	**4**	**8**	**2**	**4**	**8**
Tiny	**648**	5.22	3.42	3.51	5.77	4.09	3.68
Small	**2,540**	6.95	4.61	4.94	6.66	4.89	3.68
Med	**11,516**	8.33	5.26	5.42	8.60	5.79	4.77
Large	**25,340**	9.66	5.82	6.13	9.71	6.73	5.61
Huge	**88,748**	13.80	9.71	*	*	*	*

Table 7: Average number of nodes traversed during intersection detection for primary plus reflection rays.

Secondary Rays - Ray Trace Depth: 1, Opacity: 0.5

Traversal Time per Pixel (µs) - Opacity: 0.5

Model		Top-Down (branching factor)			Bottom-Up (branching factor)		
Name	**Size**	**2**	**4**	**8**	**2**	**4**	**8**
Tiny	**648**	11.69	10.49	13.49	12.91	11.47	14.47
Small	**2,540**	22.38	16.80	21.45	21.52	21.73	23.56
Med	**11,516**	56.98	49.17	64.68	77.99	79.30	83.27
Large	**25,340**	148.88	158.32	193.21	139.69	148.23	159.35
Huge	**88,748**	1,067.58	1,387.17	*	*	*	*

Table 8: Average BVH traversal times for primary plus opacity rays.

Nodes Traversed per Pixel - Opacity: 0.5

Model		Top-Down (branching factor)			Bottom-Up (branching factor)		
Name	**Size**	**2**	**4**	**8**	**2**	**4**	**8**
Tiny	**648**	5.12	3.33	3.50	5.75	4.03	3.74
Small	**2,540**	6.97	4.65	4.78	6.62	4.91	3.70
Med	**11,516**	8.32	5.25	5.43	8.60	5.77	4.75
Large	**25,340**	9.65	5.87	5.97	9.70	6.91	5.98
Huge	**88,748**	12.78	9.70	*	*	*	*

Table 9: Average number of nodes traversed during intersection detection for primary plus opacity rays.

References

[Ben90] Mordechai Ben-Ari: “Principles of Concurrent and Distributed Programming”. 1990. Ch 16.

[Bnt75] Jon Louis Bentley: “Multidimensional binary search trees used for associative searching”. *Communications of the ACM*, Vol. 18 Issue 9, 1975. pp 509-517.

[Cad07] G. Cadet, B. Lecussan: “Coupled Use of BSP and BVH Trees in Order to Exploit Ray Bundle Performance”. *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 2007. pp 63-71.

[Dam08] H. Dammertz, A. Keller: “The edge volume heuristic - robust triangle subdivision For improved BVH performance”. *Interactive Ray Tracing, 2007. RT '08. IEEE Symposium on*, 2008. pp 155-158.

[Eri04] Christer Ericson: “Real-Time Collision Detection”. 2004. pp 236-237.

[Ern07] M. Ernst, G. Greiner: “Early Split Clipping for Bounding Volume Hierarchies”. *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 2007. pp 73-78.

[Ern08] M. Ernst, G. Greiner: “Multi Bounding Volume Hierarchies”. *Interactive Ray Tracing, 2007. RT '08. IEEE Symposium on*, 2008. pp 35-40.

[Gar08] K. Garanzha: “Efficient Clustered BVH Update Algorithm for Highly-Dynamic Models”. *Interactive Ray Tracing, 2007. RT '08. IEEE Symposium on*, 2008. pp 123-130.

[Gun07] J. Gunther, S. Popov, H.-P. Seidel, P. Slusallek: “Realtime Ray Tracing on GPU with BVH-Based Packet Traversal”. *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 2007. 113-118

[Gut84] Antonin Guttman: R-trees: “A Dynamic Index Structure for Spatial Searching”. *ACM SIGMOD Record*, Vol. 14 Issue 2, 1984. pp 47-57.

[Hav04] Herman J. Haverkort: “Results on Geometric Networks and Data Structures”. 2004.

[Kar13] T. Karras, T. Aila: “Fast Parallel Construction of High-Quality Bounding Volume Hierarchies”. *High Performance Graphics 2013*, 2013.

[Ken08] A. Kensler: “Tree Rotations for Improving Bounding Volume Hierarchies”. *Interactive Ray Tracing, 2007. RT '08. IEEE Symposium on*, 2008. pp 73-76.

[Kno11] A. Knoll, S. Thelen, I. Wald, C.D. Hansen, H. Hagen, M.E. Papka: “Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal”. *Pacific Visualization Symposium (PacificVis), 2011 IEEE,* 2011. pp 3-10.

[Lau06] C. Lauterbach, S.-E. Yoon, D. Tuft, D. Manocha: “RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs”. *Interactive Ray Tracing 2006, IEEE Symposium on*, 2006. pp 39-46.

[Mac90] David J. MacDonald, Kellogg S. Booth: “Heuristics for Ray Tracing Using Space Subdivision”. *The Visual Computer: International Journal of Computer Graphics,* Vol. 6 Issue 3, 1990. pp 153-166.

[Mad09] F. A. Madera, A. M. Day, S. D. Laycock: “A Hybrid Bounding Volume Algorithm to Detect Collisions between Deformable Objects”. *Advances in Computer-Human Interactions, 2009. ACHI '09. Second International Conferences on*, 2009. pp 136-141.

[Nav13] Paul Navrátil, Hank Childs, Charles Hansen, Allen Malony, Carson Brownlee, and Aaron Knoll: “Collaborative Research : S12-SSI : A Comprehensive Ray Tracing Framework for Visualization in Distributed-Memory Parallel Environments”. 2013.

[Shu13] Shuai Zhao, Yue Cao, Yuxiao Guo, Si Chen, Leiting Chen: “A fast spatial partition method in bounding volume hierarchy”. *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*, 2013. pp 15-18.

[Tae10] Tao-Joon Kim, Bochang Moon, Duksu Kim, S.-E. Yoon: “RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies”. *Visualization and Computer Graphics, IEEE Transactions on*, Vol 16, Issue 2, 2010. pp 273-286.

[Wac07] C. Wachter, A. Keller: “Terminating Spatial Hierarchies by A Priori Bounding Memory”. *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 2007. pp 41-46.

[Wal07] I. Wald: “On Fast Construction of SAH-based Bounding Volume Hierarchies”. *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 2007. pp 33-40.

[Wal08] I. Wald, C. Benthin, S. Boulos: “Getting rid of packets - Efficient SIMD Single-Ray Traversal Using Multi-Branching BVHs”. *Interactive Ray Tracing, 2007. RT '08. IEEE Symposium on*, 2008. pp 49-57.

[Wal14] I. Wald, S. Woop, C. Benthin, G. S. Johnson, M. Ernst: “Embree: A Kernel Framework for Efficient CPU Ray Tracing”. *ACM Transactions on Graphics (TOG)*, 2014.

[Wei08] Wei Zhao, Rui-pu Tan, Wen-Hui Li: “Parallel Collision Detection Algorithm Based on Mixed BVH and OpenMP”. *System Simulation and Scientific Computing, 2008. ICSC 2008. Asia Simulation Conference - 7th International Conference on*, 2008, pp 786-792.

[Wtr08] Bruce Walter, Kavita Bala, Milind Kulkarni, Keshav Pingali: “Fast Agglomerative Clustering for Rendering”. *Interactive Ray Tracing (RT), IEEE Symposium on.* 2008.

[Wu13] Wu Zhefu, Yu Hong, Chen Bin: “Divide and Conquer Ray Tracing Algorithm Based on BVH Partition”. *Virtual Reality and Visualization (ICVRV), 2013 International Conference on*, 2013. pp 49-55.