EUROGRAPHICS 2021
H. Rushmeier and K. Bühler
(Guest Editors)

Volume 40 (2021), Number 2
STAR – State of The Art Report

A Survey on Bounding Volume Hierarchies for Ray Tracing

[†]Daniel Meister[1‡] [†]Shinji Ogaki[2] Carsten Benthin[3] Michael J. Doyle[3] Michael Guthe[4] Jiří Bittner[5]

[1]The University of Tokyo [2]ZOZO Research [3]Intel Corporation [4]University of Bayreuth [5]Czech Technical University in Prague

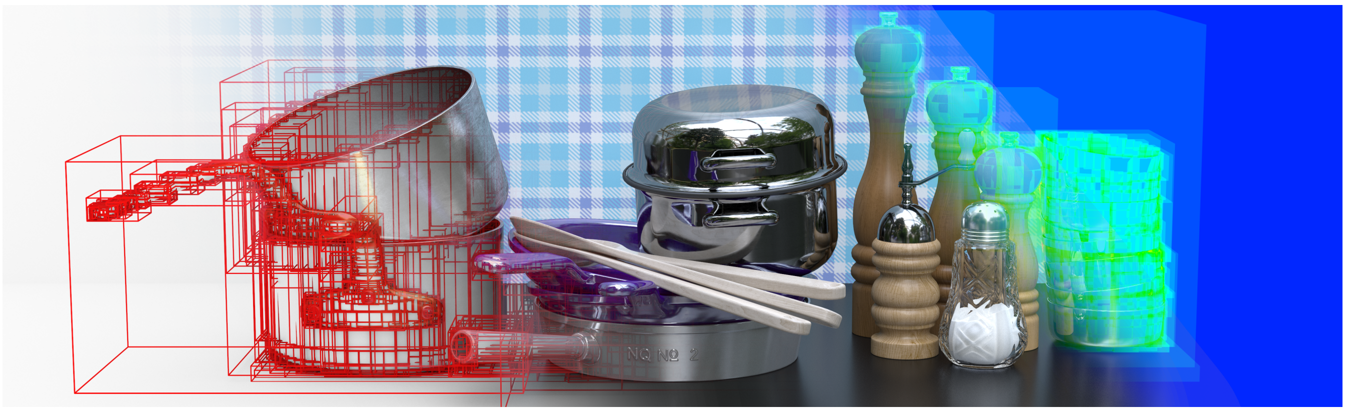

Figure 1: *Bounding volume hierarchies (BVHs) are the ray tracing acceleration data structure of choice in many state of the art rendering applications. The figure shows a ray-traced scene, with a visualization of the otherwise hidden structure of the BVH (left), and a visualization of the success of the BVH in reducing ray intersection operations (right).*

Abstract

Ray tracing is an inherent part of photorealistic image synthesis algorithms. The problem of ray tracing is to find the nearest intersection with a given ray and scene. Although this geometric operation is relatively simple, in practice, we have to evaluate billions of such operations as the scene consists of millions of primitives, and the image synthesis algorithms require a high number of samples to provide a plausible result. Thus, scene primitives are commonly arranged in spatial data structures to accelerate the search. In the last two decades, the bounding volume hierarchy (BVH) has become the de facto standard acceleration data structure for ray tracing-based rendering algorithms in offline and recently also in real-time applications. In this report, we review the basic principles of bounding volume hierarchies as well as advanced state of the art methods with a focus on the construction and traversal. Furthermore, we discuss industrial frameworks, specialized hardware architectures, other applications of bounding volume hierarchies, best practices, and related open problems.

CCS Concepts

• Computing methodologies $\rightarrow$ ***Ray tracing; Visibility; Massively parallel algorithms; • Theory of computation*** $\rightarrow$ ***Computational geometry; Massively parallel algorithms; Sorting and searching;***

1. Introduction

Ray tracing is a well-known method used for solving various problems in computer graphics [App68, Whi80]. It plays its most prominent role in realistic image synthesis, which simulates light transport based on laws of physics to achieve a high degree of realism. In computer graphics, we usually use a model of geometric physics, assuming that light travels instantaneously through the medium in straight lines [DBBS06]. Ray tracing serves as an underlying engine for finding the nearest intersections with scene primitives, which correspond to light path vertices in global illumina-

[†] Joint first authors
[‡] JSPS International Research Fellow

tion algorithms such as path tracing [Kaj86]. The problem is that a large number of rays must be traced to get plausible results. Otherwise, the resulting images suffer from high-frequency noise. The light transport simulation can be accelerated by efficient sampling techniques, by denoising algorithms, or by improving ray tracing itself by arranging scene primitives into an efficient spatial data structure. For the latter case, one of the most popular acceleration data structures for ray tracing is the bounding volume hierarchy (BVH) [Cla76].

In this report, we provide a coherent survey on bounding volume hierarchies for ray tracing. In particular, we provide the following contributions:

- To be self-contained, we start from basic principles, which might be useful for a broader audience such as game developers that are familiar with rasterization-based rendering but not completely with ray tracing (Sections 2 and 3).
- We present a collection of state of the art algorithms with a focus on construction and traversal (Sections 4, 5, and 6).
- We provide a comprehensive overview of specialized hardware for both construction and traversal (Sections 4.8 and 6.7).
- We present an overview of industrial ray tracing frameworks and renderers using bounding volume hierarchies (Section 7).
- We briefly discuss other applications of bounding volume hierarchies beyond ray tracing (Section 8).
- We provide a condensed outline of the most common techniques by way of best practice recommendations (Section 9).
- We conclude the paper by listing open problems related to bounding volume hierarchies (Section 10).

2. Preliminaries

The basic geometric operation in ray tracing is finding the *nearest intersection* with the scene for a given ray. Alternatively, we may want to find *all intersections* of the ray with the scene, or *any intersection* within a given distance along the ray. The *ray* is a semi-infinite line defined by its *origin* and *direction*. Naïvely, we can compute the ray/scene intersections by testing all scene primitives, which is prohibitively expensive as contemporary scenes consist of millions of primitives. In practice, we arrange scene primitives into various spatial data structures, which exploit the geometric proximity of the ray and scene primitives to efficiently prune the search. In the last decade, the *bounding volume hierarchy* (BVH) has become the most popular acceleration data structure for ray tracing.

The BVH is a rooted tree with arbitrary branching factor (denoted by k throughout this report) with child pointers in the interior nodes and references to scene primitives in the leaves. Each node contains a bounding volume tightly enclosing geometry in the leaves. Traditionally, binary BVHs have been used, but recently BVHs with a higher branching factor have become popular. In the context of ray tracing, *axis-aligned bounding boxes* (AABBs) are used almost exclusively as bounding volumes. Nonetheless, for some specific cases, *oriented bounding boxes* (OBB) or *bounding spheres* might also be an option. The BVH can theoretically be generalized into any number of dimensions. Nonetheless, in rendering, scene primitives are 3D entities, and thus, unless stated otherwise, we suppose that all BVHs throughout this report are in 3D.

BVHs have become popular for ray tracing thanks to the following reasons:

Predictable memory footprint The memory complexity is bounded by the number of scene primitives since each primitive is typically referenced only once. In such a case, the BVH contains at most $2n-1$ nodes (n is the number of scene primitives), which corresponds to a binary BVH with one primitive per leaf. If *spatial splits* are used, primitives can be referenced multiple times, but we can still control the number of references to a certain extent (as discussed in Section 5.1).

Robust and efficient query The BVH is robust to any scene primitive distribution thanks to its hierarchical structure. For instance, it can handle the teapot in the stadium problem. Using a BVH, we can efficiently prune branches that do not intersect a given ray, and thus reduce the time complexity from linear to logarithmic on average. BVH traversal algorithms typically have a small memory footprint, compact traversal state, and yield themselves to efficient parallelization. As a result, the BVH provides excellent ray tracing performance. In most cases, the performance of the BVH is at least comparable to the KD-tree [VHB16], which was previously considered the best data structure for ray tracing [Hav00].

Scalable construction There are various BVH construction algorithms, ranging from very fast algorithms to complex algorithms which provide highly optimized BVHs. We can thus balance the trade-off between construction speed and BVH quality. BVH quality corresponds to ray tracing speed in millions of rays cast per second (MRays/s). We typically aim to optimize the total time to compute the image, which is the sum of the construction and rendering times. In real-time applications, there is a very limited budget for both of these phases, and thus only a medium quality BVH is typically constructed, and only a very limited number of rays can be cast (e.g., a few million rays) [WIK*06]. In offline rendering, the budget is larger, and so it is prudent to invest more time in construction to produce a higher quality BVH. The additional time spent in the construction phase will be amortized by a larger number of rays traced at higher speed, yielding a net performance benefit. An overview of the time-to-image concept is shown in Figure 2.

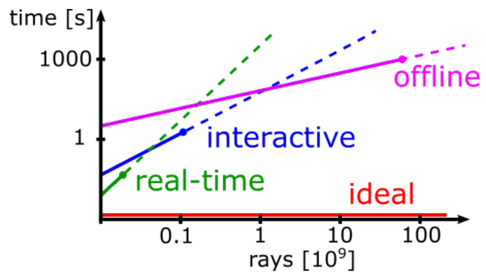

Figure 2: *An illustration of the trade-off between construction time and BVH quality for different use cases. The y-axis corresponds to time-to-image, the offset of the curves on the y-axis corresponds to the BVH construction time, the slope of the curves corresponds to the quality of the BVH, i.e., ray tracing speed. The length of the curves corresponds to typical per-frame budgets for the depicted use cases.*

Dynamic geometry Since fast BVH construction methods are available, BVHs are suitable for use with dynamic geometry. Alternatively, we can simply reflect the geometric changes by refitting the bounding volume(s) of each node, which is not possible with spatial subdivisions such as the KD-tree.

A BVH can be constructed in a top-down manner by recursively splitting the set of scene primitives into subsets. The pseudocode of the construction is shown in Algorithm 1.

```
root contains all scene primitives
push root onto the stack
while stack is not empty do
    pop node from the top of the stack
    if termination criteria for node are met then
        make node leaf
    else
        split primitives in node into children
        assign children as child nodes of node
        foreach child in node do
            push child onto the stack
```

Algorithm 1: *The basic top-down construction algorithm.*

To find the nearest intersection, the BVH can be traversed in a top-down manner starting from the root. Usually, the traversal uses a stack to store interior nodes that may potentially contain the nearest intersection. At the beginning, we push the root onto the stack. In each iteration, we pop the node from the top of the stack and compute the intersection with its bounding volume. If the ray hits the bounding volume, then we either push the node's children onto the stack in the case of a interior node, or test scene primitives in the case of a leaf node. We store the distance to the nearest intersection found so far. Using this distance, we can skip the nodes that are further than the previously found intersection. We continue the traversal until the stack is empty. For an occlusion test, when we want to know only whether the point is visible or occluded, we can thus use an early exit after finding the first intersection. The traversal algorithm is illustrated in Figure 3 and its pseudocode is shown in Algorithm 2.

```
push root onto the stack
while stack is not empty do
    pop node from the top of the stack
    if ray intersects node then
        if node is not leaf then
            foreach child in node do
                push child onto the stack
        else
            foreach prim in node do
                test whether ray intersects prim
```

Algorithm 2: *The basic stack-based traversal algorithm.*

3. Cost Function

The quality of a particular BVH can be estimated in terms of the expected number of operations needed for finding the nearest intersection with a given ray. The cost of a BVH with root N is given by

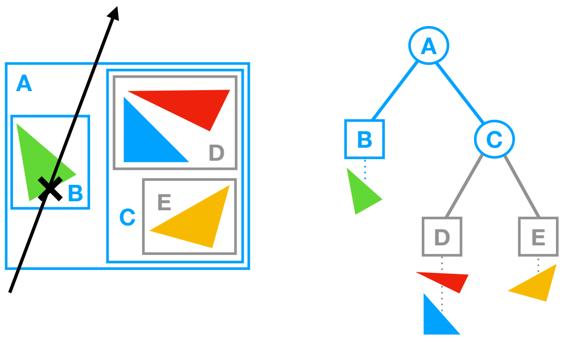

Figure 3: *An example of a BVH built over four primitives. BVH nodes contain bounding volumes (axis-aligned bounding boxes), and primitives are referenced in leaves. While traversing the BVH to find the ray/scene intersection, we cull the entire subtree of node C since the ray does not intersect the associated bounding box.*

the recurrence equation:

$$c(N) = \begin{cases} c_T + \sum_{N_c} P(N_c|N)c(N_c) & \text{if } N \text{ is interior node,} \\ c_I|N| & \text{otherwise,} \end{cases} \quad (1)$$

where $c(N)$ is the cost of a subtree with root N, N_c is a child of node N, $P(N_c|N)$ is the conditional probability of traversing node N_c when node N is hit, and $|N|$ is the number of scene primitives in a subtree with root N. Constants c_T and c_I express the average cost of a traversal step and ray-primitive intersection, respectively. These constants are usually very roughly approximated rather than expressing a precise number of assembly instructions. In practice, the ratio of these two constants is important, not the absolute values, which has an impact on the leaf sizes. For instance, it is beneficial to have larger leaves if the traversal step is significantly more expensive than the intersection test.

Using the *surface area heuristic* (SAH) [GS87, MB90], we can express the conditional probabilities as geometric probabilities, i.e., the ratio of the surface areas of a child node and the parent node:

$$P(N_c|N)^{SAH} = \frac{SA(N_c)}{SA(N)}, \quad (2)$$

where $SA(N)$ is the surface area of the bounding box of node N. By substituting Equation 2 into Equation 1, we get the following expression:

$$c(N)^{SAH} = \begin{cases} c_T + \sum_{N_c} \frac{SA(N_c)}{SA(N)} c(N_c) & \text{if } N \text{ is interior node,} \\ c_I|N| & \text{otherwise.} \end{cases} \quad (3)$$

By unrolling, we remove the recurrence:

$$c(N)^{SAH} = \frac{1}{SA(N)} \left[c_T \sum_{N_i} SA(N_i) + c_I \sum_{N_l} SA(N_l)|N_l| \right], \quad (4)$$

where N_i and N_l denote interior and leaf nodes of a subtree with root N, respectively. The problem of finding an optimal BVH is believed to be NP-hard [KA13].

The SAH assumes that ray origins are uniformly distributed outside the scene bounding box, ray directions are uniformly distributed, and rays are not occluded. These assumptions are quite unrealistic, and thus several corrections have been proposed.

Bittner and Havran [BH09] proposed the *ray distribution heuristics* (RDH), which is a method that takes into account a given ray distribution. The authors proposed to sample the ray distribution, and then use the ratio of the number of ray hits instead of surface areas:

$$P(N_c|N)^{RDH} = \frac{R(N_c)}{R(N)}, \tag{5}$$

where $R(N)$ is the number of rays hitting the bounding box of node N. Note that RDH was originally proposed for KD-trees, and its potential in the context of BVH construction has not been fully investigated.

Vinkler et al. [VHS12] proposed the *occlusion heuristic* (OH) using the ratio of the number of visible scene primitives:

$$P(N_c|N)^{OH} = \frac{O(N_c)}{O(N)}, \tag{6}$$

where $O(N)$ is the number of visible scene primitives in a subtree with root N. Using RDH or OH directly may lead to unstable results due to either undersampling or oversampling of the ray distribution. Thus, the authors proposed to blend these probabilities with the geometric probabilities given by the SAH to make the results more robust.

Fabianowski et al. [FFD09] proposed a modification for handling rays with origins inside the scene bounding box:

$$P(N_c|N)^{Inside} = \frac{V(N_c)}{V(N)} + \frac{1}{V(N)} \int_{N \setminus N_c} \frac{\alpha_{\mathbf{x}}}{4\pi} d\mathbf{x}, \tag{7}$$

where $V(N)$ is the volume of the bounding box of node N, and $\alpha_{\mathbf{x}}$ is a solid angle obtained by projecting the bounding box onto the unit sphere around $\mathbf{x}$. This expression is much harder to evaluate than the simple ratio of surface areas. Since there is no closed-form solution of the integral, the authors proposed a numerical approximation.

Aila et al. [AKL13] proposed the *end-point overlap heuristic* (EPO) motivated by the fact that most rays originate on surfaces of scene primitives, which is the case with secondary and shadow rays in global illumination algorithms. If a ray origin (or hit point) is inside multiple branches, we have to visit all of them. The idea is to penalize overlapping surfaces inside a bounding box but not necessarily in the corresponding subtree:

$$c(N)^{EPO} = \frac{1}{SA(S)} \left[c_T \sum_{N_i} SA(S_{N_i}) + c_I \sum_{N_l} SA(S_{N_l}) |N_l| \right], \tag{8}$$

where $S_N = N \cap S$ is a set of surfaces inside the bounding box of node N, and S is a set of all surfaces of scene primitives. The authors proposed to factorize the penalization of overlapping surfaces that do not belong to the particular subtree:

$$c(N)^{EPO^\star} = \frac{1}{SA(S)} \left[c_T \sum_{N_i} SA(S_{N_i} \setminus S^\star_{N_i}) + c_I \sum_{N_l} SA(S_{N_i} \setminus S^\star_{N_l}) |N_l| \right], \tag{9}$$

where $S_{N_i} \setminus S^\star_{N_i}$ is a set of surfaces that are inside the bounding box but do not belong to the corresponding subtree, and $S^\star_{N_i}$ is a set of surfaces belonging to the subtree. The authors proposed to blend this penalization with the standard geometric probabilities given by the SAH in the same manner as RDH and OH, which results in a very good correlation between this cost and the actual times. However, it is unclear how to use it directly for BVH construction. The authors also showed that the top-down construction (see Section 4.1) minimizes EPO more than other construction approaches.

4. Construction

In this section, we present a taxonomy of construction algorithms covering both basic principles and more advanced techniques.

4.1. Top-Down Construction

Top-down BVH construction was adapted from existing KD-tree construction methods [Wal07]. We start with the root node containing all scene primitives. In each step, we split scene primitives into two disjoint subsets that correspond to the node's two children, which are further processed recursively. The recursion continues until one of the termination criteria is met. The usual termination criteria are the maximum number of scene primitives in the node, maximum tree depth, or maximum memory used.

In general, there are exponentially many ways in which scene primitives can be split into two disjoint subsets. Popov et al. [PGDS09] showed that there are $\mathcal{O}(n^6)$ partitionings for axis-aligned bounding boxes (as each bounding box is defined by six planes), which is still prohibitive for any practical use. Thus, the authors proposed to use a grid approximation controlling the time complexity by the grid resolution. Note that even if we perform an optimal split in every node, it does not mean that the whole BVH will be optimal, as it is very likely to be just a local optimum of the cost function.

In practice, we split scene primitives by axis-aligned planes, similar to KD-tree construction. In the case of BVHs, each scene primitive is typically referenced only once. Note that we can relax this condition allowing spatial splits in BVH (see Section 5.1 for more details). Thus, we approximate scene primitives by a single point (e.g., a centroid of the bounding box), which always lies only on one side of the splitting plane. First, we select a splitting axis. We can test all three splitting axes and choose the best one, or we can use heuristics such as round-robin or the largest extent. Given the splitting axis, we can sample the splitting planes. There are three basic approaches for how we can split the node: spatial median split, object median split, or a split based on a cost model. The spatial median split cuts the bounding boxes in the middle. The object median split sorts scene primitives along a splitting axis, and splits them into two halves containing roughly the same number of scene primitives, which might be useful if other splits are not possible (e.g., all scenes primitives are on one side of the splitting plane or all centroids fall onto the same point). The split based on the cost model tries to minimize the cost function locally. During splitting, we cannot use the cost model directly because we do not know the cost of the children. Thus, we approximate the cost by treating the children as leaves:

$$c(N) \approx \hat{c}(N)^{SAH} = c_T + c_I \sum_{N_c} \frac{SA(N_c)}{SA(N)} |N_c|. \tag{10}$$

We can also use this approximation as a termination criterion when the cost of the node being a leaf is less than or equal to this approximation (i.e., $c_I|N| \leq \hat{c}(N)$). Popov et al. [PGDS09] proposed to penalize the overlap of child bounding boxes using an additional term in the cost function:

$$\hat{c}(N)^{OL} = c_T + \left(c_O \frac{V(\bigcap_{N_c} N_c)}{V(N)} + 1 \right) c_I \sum_{N_c} \frac{SA(N_c)}{SA(N)} |N_c|, \tag{11}$$

where c_O is a constant that controls the overlap penalty and $V(N)$ is the volume of the bounding box of node N. If there is no overlap, the cost function is simply equal to the unmodified cost function (i.e., $\hat{c}(N)^{OL} = \hat{c}(N)^{SAH}$). According to Aila et al. [AKL13], this overlap penalization is less descriptive than EPO, i.e., provides weaker correlation of the cost with the rendering times. On the other hand, unlike EPO it can be easily evaluated during the BVH construction itself.

To select the splitting plane, for the given axis, we can evaluate all $|N|-1$ splitting planes, i.e., planes between scene primitives. Evaluating all splitting planes is known as *sweeping*, and it may be costly, especially at the beginning, when nodes contain a large number of scene primitives. To address this issue, Wald et al. [Wal07] proposed an approach known as *binning*. The idea is to divide the splitting extent into b equally-spaced bins. Scene primitives are projected into these bins and then the cost function is evaluated only at the splitting planes between bins. Even for relatively small b (e.g., 16 or 32), the binning provides almost as good results as evaluating all splitting planes while accelerating the construction significantly. The number of bins can also be reduced during construction, i.e., based on the current tree depth. The full binning resolution is only required for the top of the tree, where picking the best split position matters the most. Further down the tree, a reduced number of bins provides nearly identical quality as using all bins.

Wald [Wal07] proposed horizontal and vertical parallelization of the top-down construction method using the binning algorithm. The horizontal parallelization is used for the upper levels, where only a few interior nodes contain many scene primitives. Scene primitives are equally divided between threads. Each thread projects its scene primitives into its private set of bins. After binning, the bin sets are merged, and the best splitting plane is selected. Once the number of subtrees is large enough to be assigned to all threads, the algorithm switches to vertical parallelization, where each subtree is processed by a single thread. The algorithm is designed to utilize both SIMD instructions and multithreading, and it was also extended for the MIC architecture [Wal12]. Since the size of each subtree, and therefore its construction time, can vary significantly, a task-stealing approach is essential for distributing the work efficiently across threads.

Ganestam et al. [GBDAM15] proposed a top-down construction algorithm known as *Bonsai*. The construction starts by recursively subdividing triangles into coherent clusters using spatial median splits. For each cluster, a mini-tree is built by testing all possible splitting planes to minimize the cost approximation in Equation 10. Efficiently testing all possible split positions during the construction of each mini-tree requires a fully sorted list of all possible split positions in each dimension. Ganestam et al. sort the split positions once per mini-cluster in a preprocessing step and then maintain the sorted lists during recursive construction. In the last step, the top levels of the hierarchy are built, treating mini-trees as leaves.

Hendrich et al. [HMB17] proposed a technique known as *progressive hierarchical refinement* (PHR) inspired by the build-from-hierarchy method originally proposed for KD-tree construction [HMF07]. The idea is to build an initial BVH by some method (the authors use LBVH, see Section 4.4), and then find a cut in this BVH, which is a set of nodes separating the root and the leaves. The cut is formed by nodes whose surface area is below an adaptive threshold. The cut is split into two parts by the sweeping algorithm. The threshold is refined taking into account the current depth and some nodes of the cut are replaced by their children. This procedure is applied recursively to build the whole BVH.

Wodniok and Goesele [WG16, WG17, Wod19] studied how to better approximate the cost during top-down construction, motivated by the fact that the top-down construction implicitly minimizes EPO (see Section 3). The authors proposed to construct temporary subtrees induced by a tested splitting plane to better approximate the cost. Vinkler et al. [VHBS16] introduced a parallel on-demand construction on the GPU. The core idea is to build only those parts that are visited during the traversal. Ng and Trifonov [NT03] proposed stochastic sampling of splitting planes instead of greedily taking the best one, which only minimizes the cost function locally.

Lauterbach et al. [LGS*09] proposed one of the earliest GPU-based construction algorithms using binning. Garanzha et al. [GPBG11] proposed to use uniform grids of various resolutions to accelerate binning on GPUs. Sopin et al. [SBU11] proposed another binning-based algorithm classifying nodes according to their sizes while using a different strategy for each node type. Meister and Bittner [MB16] proposed a GPU-based construction algorithm using k-means clustering instead of splitting by axis-aligned planes. Scene primitives are subdivided into k clusters using k-means clustering. Applying this procedure recursively, a k-ary BVH is built, which can be used directly, or it can be converted to a binary one by constructing intermediate levels using agglomerative clustering.

4.2. Bottom-Up Construction

An opposite approach to top-down construction is bottom-up construction by agglomerative clustering proposed by Walter et al. [WBKP08]. At the beginning, all scene primitives are considered as clusters. In each iteration, the closest pair is merged, where the distance function is the surface area of a bounding box tightly enclosing both clusters. This process is repeated until only one cluster remains. In general, agglomerative clustering produces BVHs with a lower global cost, but the construction is more time-consuming. A caveat is that the optimization is stressed in bottom levels, and thus top levels may be poorly locally optimized (unlike in top-down construction). This may be critical, as most of the traversal time is spent in the upper levels. The major bottleneck is the nearest neighbor search that has to be performed for each cluster to determine the closest cluster pair in each iteration. Walter et al. [WBKP08] proposed to use a heap and an auxiliary KD-tree. The heap stores the nearest neighbor using the distance as a priority. The KD-tree is used to accelerate the nearest neighbor search. Nonetheless, this approach is difficult to be parallelized since the topology of the KD-tree is modified by inserting and removing clusters.

Gu et al. [GHFB13] proposed a CPU-based parallel construction algorithm known as *approximate agglomerative clustering* (AAC). The idea is to restrict the search space for the nearest neighbor search by proximity given by the Morton curve (see Section 4.4). At the beginning, the scene primitives are recursively subdivided by spatial median splits based on Morton codes until each subtree contains less than a predefined number of clusters. To reduce the number of clusters in each subtree, the clusters are merged using agglomerative clustering until only a small number of clusters remains (not necessarily one). The algorithm continues to the parent, where the clusters of both children are combined; again, the clusters are merged using agglomerative clustering. This procedure is repeated until the whole tree is built. To accelerate the nearest neighbor search, the authors proposed to cache cluster distances in a distance matrix using the fact that almost all distances remain the same, and only a few are affected between iterations. Although the distance matrix requires a quadratic number of entries with respect to the number of clusters, it is feasible as each subtree contains only a small number of clusters.

AAC uses a top-down partitioning phase with a relatively large stack state (i.e., distance matrices), which is not GPU-friendly. Meister and Bittner [MB18a, Mei18] proposed a GPU-based construction algorithm known as *parallel locally-ordered clustering* (PLOC). The key observation is that the distance function obeys the non-decreasing property. In other words, it means that if two nearest neighbors mutually correspond, they can be merged immediately as there will not be any better neighbor. For parallel processing, this means that all mutually corresponding cluster pairs can be merged in parallel. Similarly to AAC, the algorithm uses the Morton curve but in a different way. The clusters are kept sorted along the Morton curve. To find the nearest neighbor, each cluster searches on both sides in the sorted cluster array, testing only a predefined number of clusters. This approach is GPU-friendly as it does not require any additional data structures such as distance matrices. The whole algorithm works in iterations consisting of three steps. First, the nearest neighbors are found using the Morton curve. Then, all mutually corresponding pairs are merged and placed into the position of the first cluster. Finally, the holes are removed using a parallel prefix scan. Usually, only a small number of iterations are needed to build the whole tree.

4.3. Incremental Construction

Incremental construction by insertion, proposed by Goldsmith and Salmon [GS87], was the first algorithmic approach for building BVHs. The idea is to start with an empty BVH and insert scene primitives one by one into that BVH. For each scene primitive, we find an appropriate leaf node by traversing the BVH from the root, and we insert the primitive into the leaf. If there are too many primitives, we split the node into two children eventually. This approach is useful if we do not know the entire input at the beginning of the construction, e.g., streaming the data through the network. However, this approach produces BVHs of lower quality in general. This issue was addressed by Bittner et al. [BHH15] proposing an efficient incremental construction algorithm. This method uses a priority queue-based approach to insert scene primitives into the BVH while greedily minimizing the cost for each insertion. To prevent reaching inefficient local minima of the BVH cost for degenerate primitive insertion orders, the method is combined with the insertion-based optimization [BHH13].

4.4. LBVH

Due to the hierarchical nature of the BVH, the parallelization of the construction is not straightforward. The BVH construction can be reduced to sorting scene primitives along the Morton curve [Mor66], where the order is given by Morton codes of fixed length (usually 32 or 64 bits). This is very convenient as many efficient parallel implementations of sorting algorithms are available, and thanks to the fixed length, the sorting can be done in $\mathcal{O}(n)$ using algorithms such as radix sort. The Morton curve is a well-known space-filling curve subdividing space into a uniform grid. Each cell of the grid is uniquely identified by a corresponding Morton code, which can be easily computed by interleaving successive bits of quantized cell coordinates. Similarly to top-down construction, scene primitives are approximated by points and projected into grid cells. The Morton curve implicitly encodes a BVH constructed by spatial median splits (see Figure 4), where the most significant bit corresponds to the topmost split and so on.

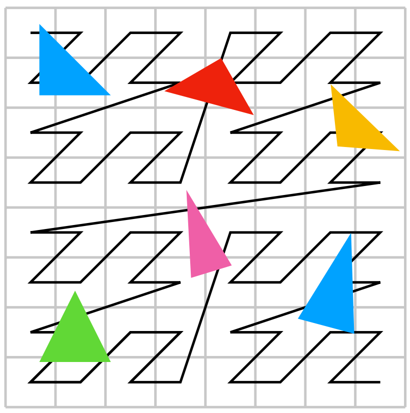
Figure 4: *An example of Morton curve in 2D using 3 bits per dimension.*

Lauterbach et al. [LGS*09] proposed a GPU-based construction algorithm known as *linear BVH* (LBVH) using the Morton curve. The BVH is built level by level in a top-down fashion, where each level requires one kernel launch. Each task splits the corresponding interval according to a given bit, eventually spawning new tasks for the next iteration. The authors proposed to use the SAH binning algorithm for lower levels to improve the BVH quality.

Pantaleoni and Luebke [PL10] combined LBVH with SAH sweeping for the upper levels, which is known as *hierarchical*

LBVH (HLBVH), and later Garanzha et al. [GPM11] replaced sweeping by binning using Morton code prefixes as bin indices. The hierarchy is constructed level by level as in the original LBVH algorithm. Nonetheless, this algorithm requires many synchronizations using atomic operations for binning.

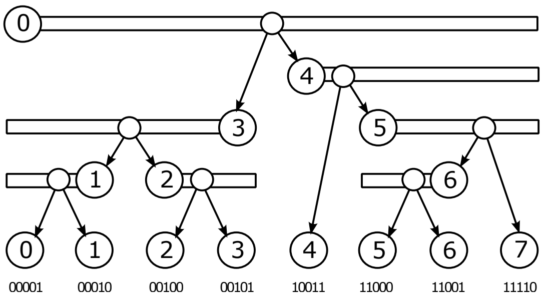

Figure 5: *An example of the compact prefix layout used in LBVH.*

LBVH was later improved by Karras [Kar12], constructing the whole tree in a single kernel launch. The idea is to use a very special node layout where interior and leaf nodes are in separate arrays. The position of each interior node coincides with one of the endpoints of the interval covering the scene primitives belonging to that node (see Figure 5). First, the second endpoint of the interval must be found. The direction (left or right) is determined by checking Morton codes of scene primitives around the first endpoint. The one with a longer common prefix of Morton codes is selected. The second endpoint is efficiently found by binary search. Given the interval, the splitting plane can be easily found by performing another binary search. The most expensive part of the construction is bounding boxes computation, which requires an additional bottom-up pass. This issue was addressed by Apetrei [Ape14], who proposed to construct both topology and bounding boxes in one pass requiring only a single kernel launch. Chitalu et al. [CDK20] combine LBVH with an ostensibly-implicit layout. Using this layout, missing parts of the complete binary tree can be efficiently determined, allowing the mapping of the implicit index representation to compact memory locations. Thanks to this layout, and by using the node index, all other attributes can be easily determined, and thus only bounding boxes must be stored explicitly (see also Section 5.5). This algorithm is the fastest construction algorithm to date. Vinkler et al. [VBH17] proposed extended Morton codes which encode not only the position but also the size of scene primitives, which may have a significant impact on the quality of resulting BVHs, especially for scenes with irregularly-sized primitives.

4.5. Topology Optimization

The problem of construction algorithms is that we cannot evaluate the cost function as the tree has not been constructed yet, and thus we only use local approximations. The idea of optimization is to build an initial BVH, and then try to improve its quality. As the BVH is already built, we can evaluate the cost function and systematically minimize it. Most of the optimization algorithms use a simplified cost model, assuming that the leaves are fixed. In this case, the cost function is reduced to the sum of surface areas of bounding boxes in interior nodes.

Kensler [Ken08] proposed using tree rotations to improve the tree quality inspired by well-known operations used for balancing binary search trees. Generally, there are four possible rotations for each interior node: swapping the left (or right) child with a grandchild in the right (or left) subtree, respectively. The basic algorithm iteratively performs the rotation providing the highest surface area reduction. The author also proposed to use simulated annealing to avoid getting stuck in local minima, i.e., the algorithm accepts rotations stochastically based on the surface area reduction, also admitting those which may increase the surface area.

Bittner et al. [BHH13] proposed a similar approach with more general operations: removing subtrees and inserting them to new positions. In this case, the search space is much larger, as any subtree can be removed and inserted into any node. Trying all possible combinations would be exhaustive. The authors proposed various strategies for choosing the order in which the subtrees are processed. For example, one strategy is to prioritize nodes with higher surface area as they might cause higher cost overhead. This approach was later improved by using Metropolis-Hastings sampling [BM15]. To search for a new position, the authors proposed to use a priority queue with a branch-and-bound algorithm using the best position found so far as a lower bound for pruning. This algorithm produces BVHs of the highest possible quality at the cost of higher build times.

Insertion-based optimization is inherently sequential. Meister and Bittner [MB18b, Mei18] reformulated this algorithm to be parallel, utilizing the computational power of contemporary GPUs. The key insight is that we do not need to remove the subtree to compute the surface area reduction. Supposing that the position for the insertion is already known, the surface area reduction can be tracked by traversing a path (in the tree) between root nodes of both subtrees. The surface area reduction is a sum of surface area differences on this path. Tracking the surface area reduction without actual removal, multiple subtrees can search for new positions in parallel. The search procedure starts from the original position. The algorithm proceeds up to the root node and visits every sibling subtree along this way while incrementally tracking the surface area reduction. A sibling subtree can be visited using a simple pre-order traversal with parent links without any additional data structure. The best position found so far is used as a lower bound for pruning, which is initially set to the original position. The actual removal and insertion may lead to race conditions, and thus synchronization is necessary. To resolve these conflicts, the authors proposed to use atomic locks prioritizing nodes with higher surface area reduction. The algorithm is about two orders of magnitude faster while providing BVHs of similar quality as the original sequential algorithm.

Karras and Aila [KA13] proposed a very fast GPU-based optimization algorithm known as *treelet restructuring* (TRBVH). The idea is to restructure treelets (i.e., small subtrees of a fixed size) in an optimal way using a dynamic programming approach. Since the algorithm proceeds level by level up to the root, treelets overlap between iterations, which allows propagating changes from the bottom up to the root leading to high-quality BVHs. This algorithm was later improved by Domingues and Pedrini [DP15] by employing agglomerative clustering instead of dynamic programming. This allows us to restructure larger treelets, and thus achieving higher quality at the same time. Note that in both cases, the authors proposed a full build strategy using LBVH as an initial BVH.

Some of the BVH construction techniques previously mentioned also contain inherent optimization phases [GHFB13, GBDAM15, HMB17]. The main difference compared to the optimization techniques discussed in this section is that they use only a single or a few optimization steps inherently connected with the rest of the algorithm.

4.6. Subtree Collapsing

Bottom-up construction and some optimization techniques produce BVHs with one primitive per leaf, which may cause not only memory overhead but also a cost increase, as it may pay off to have larger leaves (depending on c_T and c_I constants) [BHH13, KA13]. With subtree collapsing, we proceed from the leaves up to the root, comparing the cost of the node being a leaf with the actual cost of the node's subtree. If the cost as a leaf is less than or equal to the actual cost, the subtree is collapsed into a larger leaf. This technique is similar to the termination criterion based on the cost approximation. The difference is that the cost can be fully evaluated, which leads to a guaranteed cost reduction. Meister and Bittner [MB18a] proposed a GPU-based version of this postprocessing technique.

4.7. Data Layout

Contemporary processors have multi-level caches to reduce average memory access latency. The above-described metrics, such as the SAH and EPO, do not explicitly incorporate factors such as cache misses or the size of the working set. Therefore, in a real-world application, changing the order of nodes can improve cache locality and reduce traversal cost. Improving locality also helps to reduce secondary effects, including translation lookaside buffer misses and CPU-based hardware prefetching.

Yoon and Manocha [YM06] proposed a node layout algorithm known as *cache-oblivious BVH* (COLBVH) that recursively decomposes clusters of nodes and works without prior knowledge of the cache, such as the block size. In initialization, each node is assigned the probability that the node is accessed, given that the cluster's root is already accessed. The cluster with $|\mathcal{N}|$ nodes is then decomposed into $\lceil \sqrt{|\mathcal{N}|+1}+1 \rceil$ smaller ones by merging the nodes with the highest probability into a root cluster. Next, the decomposed clusters are ordered considering their spatial locality. The root cluster is placed at the beginning, and the remaining child clusters are ordered according to their original BVH positions, from left to right, in a multi-level depth-first layout. The same process is recursively applied to child clusters.

Wodniok et al. [WSWG13] proposed new layouts: *swapped subtrees* (SWST) and *treelet-based depth-first-search/breadth-first-search* (TDFS/TBFS). These layouts are determined based on the node access statistics obtained by casting a small number of sample rays in a preprocessing step. SWST aims to achieve better cache locality by swapping subtrees of a node in a depth-first layout. If the right child is more accessed than the left, the node's subtrees are exchanged. The latter, treelet-based layouts, divide a BVH into treelets by merging the most frequently accessed nodes. The difference between TDFS and TBFS is whether the treelets are created in depth-first or breadth-first order. The authors compared the proposed layouts against DFS, BFS, van Emde Boas layout, and COLBVH, showing that TDFS achieves the highest speedup on average. However, none of these layouts is always better.

Liktor and Vaidyanathan [LV16] proposed a two-level clustering scheme, which decomposes a given BVH into clusters similar to COLBVH. The key difference is the use of two different types of clusters to further reduce bandwidth and cache misses. The BVH is first recursively decomposed into a specified number of *address clusters* (ACs), in which child pointers can be represented with reduced precision (i.e., child pointers are compressed). Next, *cache clusters* (CCs) are recursively generated within each AC. CCs are cache-aware, meaning that their size is determined to fit within a cache line. They help reduce cache misses and facilitate cache boundary alignment. The AC leaves must be replaced by nodes called *glue nodes* with full-precision pointers to refer to other ACs.

4.8. Hardware Acceleration

As can be seen from the preceding sections, a great deal of progress has been achieved in the area of BVH construction algorithms. This progress has manifested as both improved construction speed, as well as improvements in the quality of the hierarchies produced. For real-time and highly dynamic content, BVH construction can still easily represent a significant portion of the rendering workload, even for the state of the art builders on high-end platforms. Motivated by this fact, a modest body of research has accumulated on specialized hardware architectures for BVH construction, which are designed to achieve superior performance and efficiency compared to software solutions.

The first published specialized architecture for the construction of acceleration structures was proposed by Doyle et al. [DFM12, DFM13]. The design is entirely fixed-function, and implements a high-quality binned SAH BVH construction algorithm. The design features an efficient arrangement of specialized circuits designed to maximize both construction throughput and efficiency. Another notable feature of Doyle et al.'s design is that it introduces a memory optimization that allows for overlapping the partitioning of one level of the BVH with the binning for the next level, leading to a significant reduction in memory bandwidth. The design achieved up to 10$\times$ the build performance of the fastest SW binned SAH builders of the era, while using much lower hardware resources. Doyle et al. also emphasized energy efficiency, and the minimization of off-chip memory accesses which expend disproportionate quantities of energy, as a key advantage that a hardware BVH build solution could offer to future heterogeneous graphics processors.

Following their earlier work, Doyle et al. [DTM17] integrated their BVH construction unit into a heterogeneous system-on-chip comprising a multi-core CPU working in tandem with a compact version of their BVH HW unit. This new design was prototyped on an FPGA. In their prototype, the upper levels of the BVH are built using the CPU, with the lower levels constructed with the accelerator. The authors demonstrate that the flexibility offered by the integrated CPU allowed the BVH build unit to be leveraged for empty space skipping in direct volume rendering, as well as in hybrid surface/volume visualization pipelines.

Following the work of Doyle et al., Viitanen et al. [VKJ*15, VKJ*17a] proposed *MergeTree*, a fixed-function HW accelerator implementing an HLBVH-style builder. This work introduces a number of innovations designed to improve the efficiency of HW builders. Similar to Doyle et al., Viitanen et al. particularly emphasize memory bandwidth and energy efficiency in their work. Working towards this goal, one notable innovation of Viitanen et al.'s design is the replacement of the radix sort of earlier HLBVH builders with a bandwidth-efficient *multi-merge sort*. The builder is capable of building BVHs in as little as two passes through the data. The main stages of the design include pre-sorting units, which feed sorted blocks of primitives to a heap-based merge sort unit, which in turn feeds into a streaming hierarchy emission unit. In addition to this, a binned SAH builder modeled after the work of Doyle et al. is included for HLBVH+SAH hybrid hierarchies. The design is capable of building hierarchies substantially faster than Doyle et al.'s design, but with the disadvantage that the hierarchies are of lower quality.

Viitanen et al. [VKJ*18] also proposed *PLOCTree*, a HW architecture which implements the parallel locally-ordered clustering (PLOC) algorithm of Meister and Bittner [MB18a] (see Section 4.2). The design also aims to minimize memory traffic, by introducing a novel streaming formulation of the PLOC algorithm which overlaps nearest neighbor calculation and merging of primitive AABBs. The major units of the design are a *sorting subsystem* that performs the initial sort of the input, and a number of PLOC *sweep pipelines*. The PLOC sweep pipelines consist of a *window memory* for storing the local windows for merging, *distance metric evaluators* and a *comparator tree* for computing distances between AABB merge candidates, and a *post-processing stage* which completes mutual nearest neighbor determination. Another innovation of this work is the use of reduced-precision arithmetic for the distance metric, which saves hardware resources. The design leads to fast builds that improve on the quality of the trees produced by MergeTree, but requires more hardware real estate to achieve this.

Recognizing the fact that many high-performance ray tracing solutions are migrating to compressed BVHs, Viitanen et al. [VKJ*17b] proposed a bottom-up update algorithm for compressed BVHs, and furthermore proposed a HW architecture for implementing this method. Viitanen et al. showed that such a HW unit could lower memory bandwidth requirements and improve performance of build and refitting algorithms which output the hierarchy in a bottom-up fashion.

The *PowerVR Wizard GPU* from Imagination Technologies reportedly features a HW-based BVH builder called the *scene hierarchy generator* [McC14]. Limited information found in public presentations reveals that it is based on a streaming voxelization of the triangles which are then spatially organized using an octree "scaffolding" and an associated 3D cache. The hierarchy is reportedly built in a single pass and in a bottom-up fashion. The design represents an interesting approach to the problem, but to the authors' knowledge, no publicly available data are available to compare its performance or BVH quality to other existing HW builders.

Aside from BVH construction, other notable works include the *HART* mobile ray tracing GPU [NKP*15], which features a hardware accelerated BVH refit unit, while relying on a CPU for the initial construction of the BVH. Woop included a HW refitting unit for B-KD trees in the *DRPU* architecture [Woo06]. The *RayCore* architecture includes a KD-tree builder which includes both binning and sorting-based build units [NKK*14]. Liu et al. [LDNL15] proposed a LBVH-style HW builder for KD-trees. The *RayTree* IP from SiliconArts offers a hardware KD-tree build solution [Sil20]. Finally, Deng et al. [DNL*17] also provide further detail on some of the works mentioned here.

5. Extensions

In this section, we introduce more general models such as wide BVHs and spatial splits. We also present more specialized techniques customized for handling dynamic geometry or non-polygonal primitives. Last, we describe various data representations for BVHs which significantly reduce memory footprint.

5.1. Spatial Splits

BVHs often adapt poorly to scenes with overlapping primitives of non-uniform sizes, which are difficult to separate by definition. On the other hand, spatial subdivisions such as the KD-tree excel in such scenarios as they subdivide space into disjoint cells while splitting the scene primitives. The drawbacks of this include higher memory consumption, slower construction, and more complicated traversal. Similarly, we can relax the restriction that each scene primitive is referenced only once in a given BVH, which brings another degree of freedom to the construction. In other words, we can make bounding boxes tighter at the cost of more references (see Figure 6). The traversal and the structure itself remains the same. A caveat is that this is not suitable for animated scenes since the refitting destroys the spatial splits.

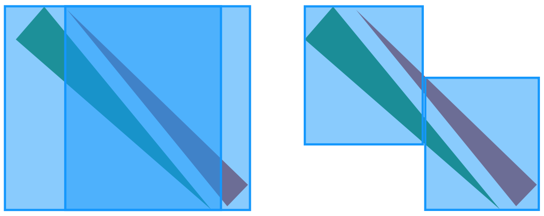

Figure 6: *The difference between a traditional BVH (left) and a BVH with spatial splits (right). Admitting spatial splits, we can achieve tighter bounding boxes at the cost of more references.*

Ernst and Greiner [EG07], Dammertz and Keller [DK08], and Karras and Aila [KA13] proposed a method known as *presplitting*. The idea is to cover large scene primitives by multiple smaller bounding boxes prior to the actual construction. Then, the bounding boxes are fed into an arbitrary construction algorithm. This process can be easily parallelized as each scene primitive is processed individually. A drawback is that the ray tracing improvement is not so significant as each primitive is processed independently not taking into account overlap with other primitives. Another drawback is that pre-splitting requires more memory, as the up-front heuristic used leads to more splitting than is actually required.

Therefore, Stich et al. [SFD09] and Popov et al. [PGDS09] proposed to allow spatial splits during top-down construction, similarly to KD-tree construction [Hav00, HHS06]. Unlike in the case of KD-trees, the cost function is piecewise quadratic since bounding boxes adapt in all three axes. Therefore, the authors proposed a modified version of the binning algorithm [Wal07]. During the construction, both spatial splits and (standard) object splits are evaluated, choosing the one minimizing the cost approximation. This approach leads to higher ray tracing performance since it takes into account the proximity of scene primitives.

Fuetterling et al. [FLPE16] proposed a parallel top-down construction with spatial splits conceptually similar to the horizontal and vertical parallelization proposed by Wald and Havran [WH06]. Similarly, Ganestam and Doggett [GD16] proposed an extension of the Bonsai algorithm to support spatial splits. Hendrich et al. [HMB17] proposed evaluating spatial splits using nodes from a cut of the hierarchy, which has only a minor influence on the build time. The number of references can be controlled by using both object and spatial splits, which is not the case with KD-trees. Thus, the memory footprint is still predictable even while allowing spatial splits. This approach provides significant acceleration, but even the parallel version is still relatively slow. in comparison with presplitting.

In both cases, we can either actually split triangles into multiple smaller triangles (using tessellation) [EG07, DK08, KA13], or just cover them with multiple smaller bounding boxes and keep triangles as they are [SFD09, HMB17].

5.2. Wide BVH

BVHs with higher branching factors are known as *wide BVHs* or *multi BVHs* (MBVHs). Wide BVHs with branching factor k are sometimes also denoted by BVHk (e.g., BVH4 or BVH8). Using wide BVHs, we can utilize parallel computing resources such as SIMD/SIMT units more efficiently by testing one ray against multiple bounding volumes during the traversal [WBB08,EG08,LSS18]. Moreover, wide BVHs are also memory efficient as they contain significantly fewer interior nodes than binary BVHs.

Unlike binary BVHs, wide BVHs have to deal with empty slots since it is very unlikely that each interior node has the maximum number of children, which typically happens around the leaf level of the tree. This generalization brings another degree of freedom in the construction. Memory efficiency highly depends on the number of empty slots. For example, a full k-ary BVH contains only $\frac{|\mathcal{N}_l|-1}{k-1}$ interior nodes, where $|\mathcal{N}_l|$ is the number of leaves, not considering spatial splits (see Section 5.1).

There are two classes of algorithms for building wide BVHs. The first class relies on an already existing binary BVH which is converted to a wide BVH by discarding interior nodes. The second class directly builds a wide BVH during construction.

In terms of the first class, Wald et al. [WBB08] proposed a collapsing algorithm trying to minimize the cost function. First, subtrees are collapsed in the bottom levels into large leaves to try to achieve an appropriate number of primitives (ideally a multiple of the SIMD width). Then, the collapsing of interior nodes is performed starting from the root and recursively processing children. The authors use three operations to minimize the cost function: merging the child node into the parent node, merging two leaf nodes, and merging two interior nodes. Concurrently, similar approaches based on collapsing were proposed by Ernst and Greiner [EG08], and Dammertz et al. [DHK08].

Pinto [Pin10] proposed a dynamic programming approach to convert a binary BVH in an optimal way. This approach was later improved by Ylitie et al. [YKL17], optimizing both leaf nodes and the interior structure at the same time. Nonetheless, as was pointed out by Wald et al. [WBB08] and Aila et al. [AKL13], the cost function may not correlate perfectly with the actual time depending on a particular traversal algorithm and the underlying hardware.

Gu et al. [GHB15] proposed *ray specialized contraction*, which collapses a binary BVH into a wide BVH taking into account the actual ray distribution. First, a binary BVH is built and rendering is performed using 0.1-0.5% of the total ray budget as sample rays to collect statistics. After recording how many rays visit each node to estimate $P(N_c|N)$, nodes with higher probabilities are collapsed first. Note that the sample rays are also used to generate the final image and are not wasted.

Although collapsing is typically easier to implement, having both binary and wide BVHs may cause memory consumption issues. If the given BVH build algorithm allows for the direct construction of wide BVHs, the conversion step approach is unnecessary. Embree [WWB*14], for example, builds wide BVHs directly using a recursive binned SAH builder. The main difference to building a binary BVH is that it requires $k-1$ splits to fill a wide node. For a partly filled wide BVH node, the child with the maximum surface area is selected to be the next splitting candidate. This heuristic of selecting the child with the maximum surface area first reduces the surface area of all of the node's children as quickly as possible. If the BVH node is completely filled or no more splits are possible, the build process continues with all valid children in a recursive manner.

The problem of the wide BVH is that it may contain too many empty slots. Fuetterling et al. [FLPE15] detached the k bounding boxes from a wide BVH node and interleaved the bounding boxes with the rest of the cluster information, including child pointers. By limiting the number of node clusters to two or four, they achieved a 5% memory reduction and marginal speedup. Merging two or more nodes with empty slots further improves memory efficiency [Oga16]. However, each node needs a child bitmask to know which nodes are its children, and more memory is needed for the bitmask as the branching factor increases.

5.3. Dynamic Geometry

Recently, ray tracing has become attractive also for interactive and real-time applications, which brings a whole family of new problems.

5.3.1. Animated Scenes

The main problem is dynamic geometry, which may change in each frame, invalidating precomputed data structures such as a BVH.

There are two main approaches for dealing with these changes. We can either reconstruct the BVH from scratch or simply refit axis-aligned bounding boxes in a bottom-up fashion (not possible for KD-trees). Note that refitting is simple for axis-aligned bounding boxes as we can compute a bounding box of a parent as a union of child bounding boxes. The resulting bounding box is the same as the bounding box computed from the geometry stored in the sub-trees. This is not the case, for example, for bounding spheres. In this case, the refitting generally leads to a gradual overestimation of the bounding volumes towards the higher nodes in the hierarchy. Reconstruction is robust to any change, but it might be too expensive and wasteful to not take temporal coherence into account. On the other hand, refitting is very fast, but the BVH may degenerate if the changes are significant. We can also do something in between such as partial updates.

Yoon et al. [YCM07] and Kopta et al. [KIS*12] proposed to use online tree rotations to reflect geometric changes. Ize et al. [IWP07] and Wald et al. [WIP08] proposed to use an asynchronous reconstruction concurrently with rendering to keep stable framerates.

Lauterbach et al. [LYTM06] proposed a heuristic to express the degree to which the hierarchy has become degraded. A key observation is that degradation is proportional to the ratio of the parent's surface area and the sum of surface areas of child nodes. Assuming that this ratio is good at the time of construction, and will worsen during the animation, the degradation for the whole BVH is defined as summed differences of the current ratios and the original ratio divided by the number of interior nodes:

$$d = \frac{1}{|\mathcal{N}_i|} \sum_{N_i} \left[\frac{SA_t(N_i)}{\sum_{N_c} SA_t(N_c)} - \frac{SA_0(N_i)}{\sum_{N_c} SA_0(N_c)} \right], \qquad (12)$$

where $|\mathcal{N}_i|$ is the number of interior nodes. Division by the number of interior nodes makes the value d independent of a particular scene. Using this value, it can be determined whether the BVH should be reconstructed or just updated. According to the authors, the hierarchy should be reconstructed if $d > 0.4$.

Bittner and Meister [BM15] proposed an optimization method for animated scenes. The idea is to optimize a single BVH for the whole animation. The authors proposed a cost function expressing the cost of the animation. The cost function is defined as a weighted mean of costs of representatives frames:

$$c_t(N) = \frac{1}{SA_t(N)} \left[c_T \sum_{N_i} SA_t(N_i) + c_I \sum_{N_l} SA_t(N_l)|N_l| \right], \qquad (13)$$

$$\widetilde{c}(N) = \frac{\sum_i w_i c_i(N)}{\sum_i w_i}, \qquad (14)$$

where $SA_t(N)$ is the surface area of node N in time t and w_i is the weight of animation frame i. This cost function can be plugged into any optimization algorithm such as the insertion-based optimization [BHH13]. The method provides good results for scenes with complex animations. One limitation is that we need to know at least a few representatives frames a priori to perform the optimization. Another limitation is that a single BVH might not be sufficient to cope with all geometric changes of more complex animations.

5.3.2. Motion Blur

Early approaches for handling motion blur in ray tracing restricted the number of time steps to two, which allowed for using linear interpolation of vertices and bounding volumes [CFLB06, HKL10, HQL*10]. Linear motion blur is not suitable for movie production rendering because it does not provide an adequate approximation for fast deforming or rotating objects. These cases require multi-segment motion blur, where different objects are assigned a different number of time steps. Gruenschlos et al. [GSNK11] used a single BVH with spatial splits (see Section 5.1) for multi-segment motion blur, where each node stores the maximum number of bounding volumes corresponding to the maximum number of time steps required. Woop et al. [WAB17] proposed a *spatial-temporal BVH* (STBVH) which efficiently supports multi-segment motion blur by adaptively performing spatial and temporal splits during BVH top-down construction, thereby lifting the restriction to always use the maximum number of time steps per BVH node.

5.3.3. Two-level Hierarchy

Describing a dynamic scene as a simple two-level hierarchy is a good fit for many rendering applications, in particular when most dynamic animation comes from rigid body transformation. Wald et al. [WBS03] proposed a two-level hierarchy where a separate bottom-level BVH is first built for each object in the scene, and then a single top-level BVH is built over all objects. Typically, the leaves of the top-level BVH store references to the corresponding bottom-level objects. If the geometry of a single object changes, now only its BVH and the top-level BVH must be updated. This has a significantly lower cost than rebuilding all BVHs or building a single BVH over the entire set of geometry. If the object animation is described as a rigid body transformation, instead of transforming the geometry itself, we can transform the ray inversely when entering the bottom-level BVH. This approach requires storing the object transformation in the top-level leaves.

The efficiency of the two-level hierarchy heavily depends on the quality of the top-level BVH. Largely overlapping objects (in world space), which are common in real-world scenarios, can quickly reduce the culling efficiency of the top-level BVH, as a ray intersecting the overlapping region will need to sequentially intersect all overlapping objects. For reducing the overlap, Benthin et al. [BWWA17] proposed to apply partial re-braiding. After building an individual BVH for each object, the approach opens and merges bottom-level BVHs during top-level BVH build. The opening allows for finding better splitting planes during top-level construction, thereby reducing overlap and increasing SAH quality. Excessive opening of bottom-level BVHs is avoided by only applying this step where it would provide the most gain in the terms of the SAH quality. A general consequence of the re-braiding step is that the top-level BVH will contain multiple entry points to the same object.

DirectX Ray Tracing supports a two-level acceleration structure. Lee et al. [WJLV19] extended the DirectX programming model by introducing a *programmable instance* (PI) that can be referenced by both the top-level and bottom-level BVHs. When a ray intersects a PI, a traversal shader is invoked, and traversal is redirected to a different acceleration structure. This arbitrary acceleration structure selection enables procedural multi-level instancing and stochastic LOD selection that reduces memory bandwidth. Furthermore, the traversal shader can simplify the implementation of lazy build [LL20], which usually requires complex ray scheduling. *Multi-pass lazy build* (MPLB) is an iterative algorithm consisting of two parts: a pass to find the unconstructed visible bottom-level BVHs by dispatching a batch of rays and another pass to construct them. MPLB dramatically reduces construction costs, especially in dynamic scenes. The number of iterations can be reduced by prebuilding bottom-level BVHs which have been determined by rasterization to be directly visible and also those which have been traversed in the previous frame.

5.4. Non-Polygonal Objects

Hair and Fur AABBs do not tightly fit long, thin, curved, or diagonal primitives. In addition, the overlap between AABBs of neighboring primitives can be considerable, and thus rendering hair and fur is expensive. *Oriented bounding boxes* (OBBs) can enclose such objects more tightly in exchange for increased storage and traversal cost. Woop et al. [WBW*14] showed that using both AABBs and OBBs reduces the number of traversal steps and intersection tests while avoiding a large increase in memory consumption. To partition primitives during construction, five different splitting strategies are performed choosing one with the smallest SAH cost: (1) object and (2) spatial splitting in world space, (3) object and (4) spatial splitting in a coordinate frame aligned to the orientation of hair segments, and (5) clustering hair segments of similar orientation. Therefore, the construction is slow, making it challenging to handle motion blur and dynamic objects.

Using RTX-enabled hardware, Wald et al. [WMZ*20] achieved significant speedup by performing OBB tests at the primitive level only, without a complicated construction method. By treating each primitive as a single instance, the OBB tests can be efficiently performed by hardware supported instance transforms. Thus, the expensive context switch between the hardware BVH traversal and intersection shader can be avoided. However, each primitive requires more information, including an affine transform matrix, which leads to higher memory consumption.

Traversal costs are not the only problem in rendering thin primitives. Primitives thinner than a pixel can cause aliasing without a high sampling rate. This issue can be avoided by increasing the radius of spheres, cylinders, or curves based on the distance from the viewpoint, and compensating for this by making them transparent [GIF*18]. When using this technique, each node must also be expanded to encompass the enlarged primitives. The increased cost can be mitigated by stochastic transparency [ESSL10, LK11].

Metaballs Gourmel et al. [GPB*09, GPP*10] proposed a BVH construction method suitable for metaballs. They build a BVH over metaballs' bounding spheres using spatial splitting. The radius of a bounding sphere is equivalent to the maximum influence range of the metaball inside, and each leaf node in the resulting tree contains all the split metaballs needed to compute isosurfaces that overlap with itself. The box that encompasses the bounding spheres does not necessarily tightly fit the resulting isosurface. However, one can make it tighter by precomputing the upper bound of the range that each metaball can affect. It is also helpful to remove metaballs that contribute only to the isosurface that does not intersect the node to which they belong.

5.5. Compact Representation

A BVH node contains information such as child node pointers, the number of leaves, and bounding box(es). The memory consumption of a BVH becomes enormous as a scene size grows. This can be addressed by increasing the branching factor, compressing geometric data such as bounding boxes and vertex coordinates with reduced precision, removing pointers to child nodes and primitives by using a complete tree, or representing mesh triangle connectivity using triangle strips. These techniques are often used in conjunction with each other.

Reduced Precision Bounding boxes account for a large fraction of the data stored in nodes. With single-precision, a node extent (minimum and maximum) consumes 24 bytes. Mahovsky and Wyvill [MW06] represented the coordinates of child nodes' bounding boxes relative to the parent using fewer bits to reduce memory overhead. The quantized box must conservatively cover the original bounding box not to undermine the results of intersection tests. There is a performance penalty due to decoding the compressed boxes and the extra ray and node intersection tests caused by the slightly loosened bounds.

Hierarchical mesh quantization (HMQ) [SE10] stores a BVH and the triangles of a scene in a single unified data structure. A high compression rate is achieved by quantizing each vertex of the triangle in a leaf node as a local offset of the leaf bounding box. However, adjacent triangles stored in different leaf nodes can create gaps. The paper addresses this issue by snapping the vertices and leaf bounding boxes to a global grid. Globally snapped bounding boxes do not need to be stored in memory because they can be easily snapped on the fly when decoding the vertices.

Ylitie et al. [YKL17] showed that compressed wide BVHs reduce memory traffic and improve performance for incoherent rays on GPUs. They quantized child node boundaries to a local grid and stored each plane with 8 bits. The origin of the local grid, i.e., the minimum of the AABB of a wide BVH node, is stored as three floating-point values without compression. The scale of each axis of the local grid can be represented by only the 8-bit exponent of the floating-point value by restricting it to a power of two. Thus the local grid itself consumes 15 bytes per node.

In wide BVHs, leaf nodes make up most nodes, but they are less often intersected than the inner nodes. Based on this insight, Benthin et al. [BWWA18] introduced dedicated compressed multi-leaf nodes and achieved significant memory reduction while minimizing performance degradation by compressing only the leaf nodes.

Compression *Random-accessible compressed BVH* (RACBVH) [KMKY10] decomposes a BVH into a set of clusters to support random access on the compressed BVH. In the RACBVH representation, bounding boxes are compressed using hierarchical quantization and triangle indices using delta coding. Inside each cluster, node connectivity is expressed by storing a parent index instead of child indices. The parent index can be compactly encoded using the position in the front maintained during compression. Furthermore, clusters and meshes are compressed using a dictionary-based compressor. Clusters that do not fit in the pre-allocated memory pool are managed based on a least recently used replacement policy. Since atomic operations are involved, this method suffers from a lack of scalability.

Parent-Plane Sharing Two child nodes share at least six planes with the parent node in a binary BVH, thus storing only six planes instead of twelve reduces memory consumption [Kar07, FD09, EW11]. A similar idea is used for other data structures. The *bounding interval hierarchy* (BIH) [WK06] and *spatial KD-tree* (SKD-tree) [HHS06] subdivide the parent bounding box into two overlapping or disjoint regions by two parallel bounding planes. The *restricted boxtree* [Zac02] and *single slab hierarchy* [EWM08] use a bounding plane for each node, and the *B-KD tree* [WMS06] and *DE-tree* [ZU06] store two pairs of axis-aligned planes at each node. The *H-tree* [HHS06] allows from one to six planes to be used per node. *Dual-split trees* [LSMY19,LVY*20] store two planes that are not necessarily parallel at each node. Using fewer bounding planes also avoids redundant intersection tests against shared planes. However, recovering the bounding box for each node requires maintaining the tree state from the root to the current node, which may offset the benefits. As the branching factor increases, fewer planes can be shared.

Triangle Connectivity Strips are a compact way of representing the connectivity of triangles. *Ray-Strips* [LYM07] use a two-level data structure. A BVH is used for the *top-level acceleration structure* (TLAS), although an arbitrary data structure other than a BVH (such as a KD-tree) can also be used. Each leaf of the TLAS stores a strip and an object hierarchy (SKD-tree) built on the strip. Strips are generated using an algorithm developed for rasterization [ESV96]. Later, Lauterbach et al. [LYTM08] improved Ray-Strips by adopting a SAH-aware strip-generating algorithm called *Strip-RT*, and by including all vertex information in the strip to reduce non-local memory accesses. Strip-RT can generate longer strips with higher spatial coherence. HMQ [SE10] compresses vertex connectivity by storing short indexed strips containing up to a predetermined number of triangles (14 in the paper) in the leaf node. In practice, all possible vertex sequences of strips (e.g., two strips of length 4 followed by two strips of length 3) are stored in a look-up table, and each leaf node only has an index into this table.

Implicit Indexing In a complete k-ary tree, the child nodes of the node i are $[ki+1, ki+k]$ and the parent node is given by $\lfloor (i-1)/k \rfloor$, supposing that the index of the root is 0. Therefore, it is not necessary to store pointers to the child nodes. Additionally, there is no need to store the number of primitives if each leaf has the same number of primitives. To create a complete tree, one must use object median splitting. As a result, the quality of the resulting tree is inferior to those which are SAH-optimized.

Cline et al. [CSE06] proposed a two-pass algorithm to construct a complete k-ary tree (k is set as 4). First, the number of objects enclosed in each node is computed in a bottom-up fashion in the first pass. Next, the BVH is built by recursively partitioning the objects using a median split variant to support arbitrary branching factors. The objects are partitioned (twice for $k = 4$) along the bounding box's longest axis so that the number of primitives after partitioning matches the pre-calculated number assigned to each node. When assuming that each leaf node has a single primitive, $k\lceil \frac{n-1}{k-1} \rceil + 1$ nodes should be allocated so that every interior node has k children.

Similarly, *minimal BVH* [BEM10] uses a complete binary tree. This paper approximates each node's bounding box by trimming the parent's bounding box by one or two parallel planes but does not store the splitting planes' actual location in the node. Instead a global parameter $\zeta \in (0,1)$ is used as a reduction factor. During construction, each node is subdivided along its longest axis. Each child node's bounding box can be reduced in volume relative to its parent node by ζ, either by cutting off the lower side, the upper side, or both sides. There is another case where no empty space reduction is possible. Therefore, each node needs two bits to cover all four of these situations. Because object median splitting is used, cutting only one side of the bounding box can cause an empty space deadlock (which can occur if there are primitives on the diagonal). Cutting both sides of the box, however, can avoid this deadlock.

Bounding boxes can be determined by the vertices of the primitives they contain. Thus, if we know which vertices span a node, we can implicitly represent the whole tree structure only with geometry, i.e., only by reordering primitives (see Figure 7). The *No-Memory BVH* [EBM12] is a representation in which each node implicitly stores bounding triangles that span its bound, and thus requires no memory because bounding boxes are no longer needed. This approach is analogous to the classic photon map [Jen01] where photons' order represents a KD-tree, and one photon lies on a splitting plane. Six bounding triangles are needed at most to determine a bounding box. However, in practical implementation, the parent's bounding box is trimmed by two parallel planes at each node similar to the B-KD tree. Therefore, each node uses only two bounding triangles.

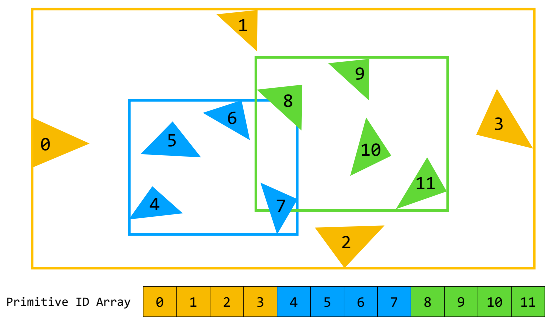

Figure 7: *No-Memory BVH represents a tree structure by the order of primitives alone. In this 2D example, four consecutive primitives determine the bounding box of each node.*

It is worth mentioning that divide-and-conquer ray tracing [Mor11,NIDN13,RN13,VBHH13,dLPP14] is also known to be a non-memory consuming method. It naturally achieves lazy construction because partitioning only occurs where rays hit, and casting a bundle of rays amortizes the construction cost. The resulting acceleration structure is not explicitly saved in memory, but a large amount of temporary memory is required to store data such as ray properties and temporarily generated bounding boxes.

Chitalu et al. [CDK20] used a perfect tree as an implicit tree. They eliminated virtual nodes and the pointers to child nodes by computing a mapping between a node's position in memory and its place in the implicit tree with simple bit manipulations (see Figure 8). The construction is done in a bottom-up manner and is very fast because there is no need to track radix key ranges and precompute the number of objects enclosed by each node. It is worth noting that it supports arbitrary arities. The number of nodes is slightly more than it should be because some interior nodes have a single child.

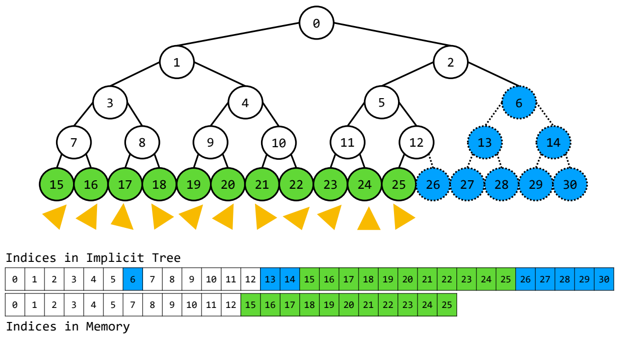

Figure 8: *An example of a binary ostensibly-implicit tree. The blue nodes are non-existent virtual nodes, and the mapping between the index in the implicit tree and the actual location in memory is performed with simple bit manipulations.*

Most of the methods discussed here do not take the SAH into account, or incur the overhead of decoding compressed boxes and slabs, which results in poor performance compared to SAH-optimized BVHs. The existing methods alleviate this issue by using coherent ray bundles to amortize the cost of decoding compressed nodes or using a two-level BVH (standard BVH nodes near the root and compressed nodes near the leaves) to balance performance and memory consumption. For scenes that are too large to fit in main memory, using a compact representation may lead to faster rendering than out-of-core techniques.

6. Traversal

The key benefit of spatial acceleration data structures for ray tracing is that they reduce the number of ray/primitive intersection tests at the cost of traversing the data structure. In this section, we describe basic principles of ray traversal as well as more advanced algorithms tailored for modern parallel architectures.

6.1. Traversal Orders

There are three main types of traversal: 1) The most common is first-hit traversal (closest hit test), which finds the nearest object in the direction of a ray from its origin. A typical use case is to estimate the radiance at a shading point by importance-sampling BRDFs in path tracing. 2) Any-hit traversal (occlusion test) is used to find any object in a specific direction from a ray origin (and within a specified range). This is useful for ambient occlusion and shadow computation. 3) Multi-hit traversal is used to find one or more primitives that intersect with a ray. It can be used to find the h closest intersections from a viewpoint or to find all intersections. It allows coplanar transparent or non-refractive surfaces to be rendered accurately and rapidly by avoiding the need to retrace a BVH from the root every time a ray hits an object.

6.1.1. First-Hit Traversal

First-hit traversal is most widely used and is indispensable for computing the radiance at a shading point. Therefore, most ray tracing engines are optimized for this type of traversal. In binary BVHs, this can be done efficiently by pushing the farther intersected node onto a stack and visiting the closer one first. On the other hand, in wide BVHs, each ray may intersect more than two nodes, and they must be visited in a front-to-back order to be efficient. There is a maximum of $k!$ orderings for a branching factor k. Two common heuristics can be used for sorting. One is the sign heuristic that determines the ordering only by the direction of a ray and the split axes used during construction [DHK08, EG08, FLP*17]. BVH4 by Dammertz et al. [DHK08] and dynamic ray stream traversal [BAM14] only supports 8 of 4!, and ordered ray stream traversal [FLPE15] supports all 4! orderings. WiVe [FLP*17] can handle more combinations and works for BVH8 and BVH16. Another is the distance heuristic, which sorts nodes by ray distance [WBB08, Áfr13, VKJT16]. When traversing a wide BVH, in most cases, the number of intersecting child nodes will be as few as 0 to 3, regardless of k. Therefore, one can speed up execution by adding a code path to sort a small number of nodes and performing full sort only if there are 4 or more hits [Áfr13, WWB*14].

6.1.2. Any-Hit Traversal

When computing soft shadows from area lights, calculating illumination from many lights, evaluating ambient occlusion, or connecting path vertices in a bi-directional algorithm, occlusion rays account for a large portion of the total rays used for rendering. When assuming non-transparent geometry, there is no need to visit intersecting nodes in the occlusion test in front-to-back order because we can immediately terminate traversal when a ray and any object intersect. Any-hit traversal can be accelerated using a different metric than the one used for radiance rays, and optionally with a dedicated BVH built with that metric.

Inspired by the SAH, Ize and Hansen [IH11] proposed the *ray termination surface area heuristic* (RTSAH) incorporating a continuous visibility function into the cost function. The key observation is that the probability of hitting individual children is not independent since we can terminate as soon as the first intersection is found. Thus, the idea is to compute the costs of individual children and traverse the one with the lowest cost first. Necessary probabilities in the cost function can be computed similarly to radiosity form factors. The authors used RTSAH just to determine the traversal order in each interior node. How to optimize the BVH construction using the RTSAH is not clear.

Surface area traversal order [NM14] (NodeSATO) accelerates tests simply by visiting the child node with the larger surface area first. The rationale is that rays are more likely to be occluded by a node with a larger surface area. The authors also proposed two other traversal orders, PrimSATO and PrimNumTO. PrimSATO prioritizes a node with a higher average or maximum surface area of each primitive in each node, and PrimNumSATO prioritizes a node with a lower number of primitives. The optimal of the three proposed methods depends on the scene. Although developed for a binary BVH, their extension to a wide BVH is trivial. Similar to RTSAH, SATO does not take into account the distribution of rays in a scene. Thus, its performance can be degraded if shadow rays do not intersect objects in large nodes.

Feltman et al. [FLF12] proposed the *shadow ray distribution heuristic* (SRDH) combining the idea of RDH [BH09] with a cost model that takes into account the traversal order of children in the same manner as RTSAH. The cost model incorporates a *traversal-order kernel function*, which expresses the probability of traversing the left child first (SRDH is limited to binary BVHs). The authors proposed a top-down construction algorithm similar to standard SAH approximation while also considering the actual ray distribution. Splitting each node is not only optimized over different partitioning options but also over multiple traversal-order kernel functions, including constant, front-to-back, or back-to-front orders.

Since it is not desirable to have an extra BVH dedicated to occlusion tests due to the initialization cost and memory overhead, Ogaki and Derouet-Jourdan [ODJ16] proposed a method to speed up shadow ray traversal of a wide BVH only by changing the order of child nodes. In this method, sample rays are cast to estimate the distribution of rays after constructing an SAH-optimized BVH, and then the children of each node are reordered so that the traversal cost is reduced. The performance of closest-hit traversal is completely unaffected if intersected child nodes are sorted in ray order.

Optimizing traversal order and reordering are conceptually the same, and neither changes the topology of a BVH. A difference is that the former chooses a node to visit next at runtime to minimize some cost (e.g., the number of intersection tests), and the latter is only done once after construction, and there is no overhead during traversal. Since estimating the ray distribution has a non-zero overhead, a large number of shadow rays must be cast to amortize the cost. If the ray tracing kernel is highly optimized, the use of sample rays may not be suitable for dynamic scenes.

6.1.3. Multi-Hit Traversal

Multi-hit traversal is a generalization of first-hit traversal, finding all intersections or the nearest h intersections along a ray. It has received significantly less attention than other traversal methods. First-hit traversal does not work well with coplanar surfaces and numerically close objects, which is particularly problematic when handling interfaces between two transparent materials such as glass and water. When reflected and refracted rays are traced with a negative ε-offset, the same intersections may be erroneously repeated. For planar objects, we can avoid the self-intersection problem by excluding the primitives from which the ray originated. However, it is not a fundamental solution as this trick does not work for non-planar primitives. If a ray is traced with a positive ε-offset, it may miss primitives to be intersected. Multi-hit traversal addresses such issues, enabling efficient and accurate rendering of transparent objects. There are two types of multi-hit traversal: those that find a limited number of intersections and those that find all intersections. In the former case, using a pre-allocated buffer that can store the required intersections is more efficient than the naïve implementation [GNK14] that adds all the intersections to a dynamic list and processes them later. When all intersections are required, it is generally not possible to predict the number of intersections, and therefore, the size of the buffer that will be required. In this case, a method that works without a dynamic list or a preallocated buffer is preferred. The same applies to situations where many intersections are required.

Gribble et al. [GNK14] proposed a buffered algorithm that can terminate traversal as soon as the h closest intersections are found. They assume that the acceleration data structure is built by spatial subdivision so that there are no overlapping nodes, and nodes are visited in a strict front-to-back order. Therefore, this method cannot be used for BVHs built with object splitting.

Amstutz et al. [AGGW15] showed that multi-hit traversal could be implemented using highly optimized ray tracing engines such as Embree [WWB*14] and OptiX [PBD*10] by using their intersection callbacks. All intersections are stored in a sizeable hit-point buffer, which is pre-allocated to avoid dynamic memory allocation during traversal. They evaluated two methods to sort intersection points. In the first approach, aiming to maximize cache locality, intersections are added to a local buffer using insertion. When the traversal is complete, the intersections are already sorted by distance. In the second approach, intersections found during traversal are stored in a local buffer without regard to the distance from the ray origin. Sorting is done after traversal is complete, and thus, SIMD utilization is improved.

A more efficient approach is to avoid unnecessary node traversal by successively narrowing the valid ray interval once a specified number of intersections are found. This technique is called node culling multi-hit BVH traversal [Gri16] and can be implemented using, for example, Embree [GWA16] and DirectX Ray Tracing [Gri19]. When several intersections are required instead of all intersections, the DXR any-hit shader implementation achieves approximately twice the naïve multi-hit traversal performance.

Wald et al.'s iterative method [WAB18] efficiently finds multiple intersections by tracking traversal state across successive queries. In contrast to spatial partitioning structures such as the KD-tree, BVHs do not guarantee that nodes will be traversed in front-to-back order. However, this can be achieved by replacing a per-ray traversal stack with a priority queue.

6.2. Numerical Issues

A robust ray-AABB intersection test [WBMS05, MCSM18] improves the robustness of BVH traversal, but it still causes false-hits and false-misses due to rounding errors. False-hits are classified as hits even though a ray does not hit a bounding box, which results in slight performance degradation. False-misses, on the other hand, mark the ray as not hit even when it hits the bounding box, and can cause a variety of problems, including holes in the object and light leaking out of gaps in the object. They also cause artifacts with non-photorealistic rendering, where object indices are used to draw contours. These problems are not solved by increasing the number of samples. Ize [Ize13] showed that false-misses could occur if the distance between the entry and exit points $(t_{far} - t_{near})$ is less than or equal to two ULPs (*unit of least precision*) by simple error analysis. He proposed an algorithm that only needs adding two ULPs to each component of the inverse ray direction for computing t_{far} before traversal.

6.3. Stream Traversal

Most ray tracing implementations on CPU or GPU make explicit or implicit use of the underlying vector units to accelerate BVH traversal. They either use the entire vector unit width to trace a single ray or trace a ray per (logical) element of the vector unit, which corresponds to processing w rays per w-wide vector unit.

Tracing more rays, or what is typically called a *ray stream*, can provide further performance benefits [GR08, Tsa09, BAM14, FLPE15, Gas16], as it allows for amortizing the access of node/primitive data and the cost of determining the traversal order for k-wide BVHs and the related stack operations over many rays. Moreover, if multiple rays follow the same control flow path, the effective vector unit utilization is increased. If rays are not following the same control path due to different traversal orders, the ray stream is essentially split into sub streams [GR08, BAM14, FLPE15], or rays not following the chosen traversal order are marked as invalid during top-down BVH traversal [Tsa09]. In case the rays in the stream can be efficiently bounded by bounding primitives such as planes or intervals, a bounding-primitive intersection test per BVH node can cull all rays in the stream in a single step [FLPE15]. A different approach proposed by Gasparian [Gas16] batches up rays at fixed points inside the BVH. All rays associated with such a point are later processed together as they likely exhibit a higher degree of coherence.

The efficiency of tracing ray streams largely depends on the amount of available ray coherence during BVH traversal and primitive intersection, as the amortization of memory access or computational cost only works when multiple rays are involved. If the rays are too incoherent so that only a single ray is active most of the time, the overhead of handling ray streams during traversal can offset the benefit.

6.4. GPU Traversal

Traversal on GPUs is especially challenging due to the non-locality of the data access and control flow when using any acceleration data structure. Furthermore, GPUs execute 32 threads in a parallel SIMD block called a *warp*. To prevent cores from stalling when a ray has been completely traversed, new rays are loaded from a global work queue using so-called persistent threads. The control flow divergence can be mitigated by breaking the traversal loop into two independent loops, i.e., iterating over the hierarchy and iterating over primitives [AL09]. Each of the two parts of the loop body can either be implemented as an *if* or *while* block. An if-if traversal leads to alternate testing of nodes and primitives if any of those threads need to process an inner or leaf node, respectively. In the case of while-while traversal, the hierarchy is traversed until all threads reach a leaf node and then they process these in parallel. As this may lead to a single thread being active in a warp, the algorithm switches to the other block if the number of waiting threads exceeds a given threshold. In addition, the traversal of inner nodes is continued in the case of while-while traversal, until a second leaf node is reached. This way, memory throughput is increased at the cost of a few additionally traversed inner nodes. Algorithm 3 shows the traversal algorithm including all proposed optimizations referred to as *persistent speculative while-while*.

```
$ray \leftarrow$ fetch_ray()
$node \leftarrow root$
$leaf \leftarrow \emptyset$
while $ray \neq \emptyset$ do
    while $node$ does not contain primitives do
        traverse to the next $node$
        if $node$ contains primitives and $leaf = \emptyset$ then
            $leaf \leftarrow node$
            traverse to the next $node$
        if number of $(leaf \neq \emptyset)$ per warp $> threshold$ then
            break
    while $leaf$ or $node$ contain untested primitives do
        perform a ray-primitive intersection test
    if $ray$ terminated then
        $ray \leftarrow$ fetch_ray()
        $node \leftarrow root$
```

Algorithm 3: *The persistent speculative while-while traversal algorithm. Speculative traversal of inner nodes is marked in blue and ray fetching for persistent threads in green.*

As shown by Guthe [Gut14], the main performance limiter of GPU traversal is latency – i.e. cores waiting for data to be loaded or stored – and not memory bandwidth or peak operations per second. Although GPUs are excellent at hiding memory latency, several strategies originally developed for latency hiding on CPUs can also be used to improve performance. One approach is to use instructions that are independent of the result from previous operations. This allows the scheduler to reorder instructions, and thus utilize the cores more efficiently. Typical candidates are independent loop iterations, which simply requires unrolling the loop. As the efficiency increases with the number of iterations, increasing the number of child nodes in the spatial hierarchy to four reduces latency at the cost of more arithmetic instructions. When sorting the intersected child nodes, a sorting network can be used instead of a traditional sorting algorithm, since that has less dependent instructions.

While the optimal hierarchy width for traditional spatial hierarchies is four, wider trees can be even more efficient when using a compressed representation [YKL17]. One of their key observations is that sorting the child nodes is prohibitively expensive for wider trees. Instead, they use an octant based pre-ordering of the child nodes and traverse them based on the signs of the ray direction vector. The traversal order is implicitly stored when constructing the hierarchy. In addition, the topmost entries of the traversal stack – including the result of the child intersection tests – are stored in shared memory to reduce memory transfers.

While many approaches seek to improve the performance for any type of rays, Lier et al. [LSS18] explicitly tackled the problem of divergent thread execution and memory access for incoherent rays. They propose to distribute the intersection tests for the child nodes or triangles to 2, 4, or 8 adjacent threads. This requires both the inner and leaf nodes to have the corresponding number of triangles. Thus, the increase in coherence is better for wider trees. While using binary trees and two cooperating threads shows no improvement over the baseline traversal algorithm [AL09], a tree width of four can result in up to a threefold speedup for highly incoherent rays. Although the divergence decreases with tree width, the optimum is four due to an increase in the number of intersection tests for wider trees.

Figure 9 shows the relative trace performance compared to primary rays using the original algorithm of Aila and Laine [AL09]. Despite improvements in GPU traversal of incoherent rays, they still require approximately twice the time of tracing primary rays.

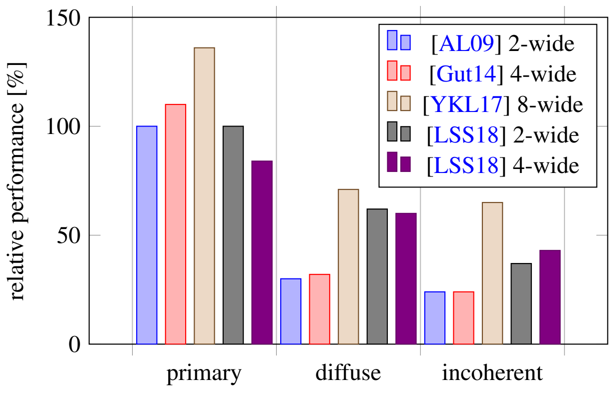

Figure 9: *Relative trace performance for different ray types. The performance of primary rays using persistent speculative while-while traversal is used as a reference.*

6.5. Stackless Traversal

It might be difficult to maintain a full stack per ray on highly parallel architectures such as GPUs because the amount of on-chip memory per HW thread is small. This fact has motivated researchers to study the possibility of using stackless traversal.

Smits [Smi98] proposed the first stackless traversal method using *skip pointers*, where traditional child links are replaced by *hit* and *miss* links. During the traversal, based on the result of the intersection, an appropriate link is selected. Using these links is simple, however, the traversal order of children is fixed. In practice, it is desirable to visit children in front-to-back order. To address this issue, Hachisuka [Hac15] extended this approach using six sets of hit and miss links for six major directions (positive and negative directions).

Laine [Lai10] adopted the restart trail technique from stackless KD-tree traversal. The idea is to traverse from the root multiple times, skipping subtrees visited in previous passes, and continuing to not yet visited subtrees. To do that, parts of the hierarchy that have already been processed are explicitly stored, using one bit per hierarchy level, while exploiting the fact that the order in which children are traversed is fixed for a given ray. The authors proposed to use a short stack with only a few entries to accelerate traversal even further. Thus, the nodes are popped from this stack until the stack is empty. Only in this case, the restart trail is performed. Vaidyanathan et al. [VWB19] generalized this approach to wide BVHs indicating how many children have already been traversed in each level.

Hapala et al. [HDW*13] proposed a stackless traversal using parent links with simple state logic indicating whether we came from the child, sibling, or parent using only two bits. A caveat is that the algorithm relies on efficiently determining the traversal order of children, which restricts the algorithm to simple heuristics such are those based on a sign of the ray direction. Barringer and Akenine-Möller [BAM13] proposed two stackless traversal algorithms for binary BVHs. The first algorithm is designed for implicit trees (see Section 5.5), exploiting the implicit layout to efficiently backtrack in the tree using two bits per hierarchy level. The second algorithm uses parent links to backtrack in general (sparse) trees storing the traversal state in one bit per hierarchy level. Unlike the approach of Hapala et al. [HDW*13], the traversal order of children is not reevaluated multiple times since it is already encoded in the traversal state, and thus it is possible to use more complex heuristics to determine the traversal order of children. Áfra and Szirmay-Kalos [ASK14] extended this approach to support wide BVHs. One disadvantage is that some interior nodes might be visited twice (on the way down and on the way up).

Binder and Keller [BK16] proposed an efficient stackless traversal with backtracking in constant time targeting contemporary GPU architectures. The idea is to keep a path to the current node encoded in a bitset. Similarly to Laine [Lai10], the trail is explicitly stored, i.e., which parts have already been visited. Based on the number of leading zeros in the trail, one knows how many levels to go up. The path to such a node can be determined using the encoded path and the number of leading zeros. In order to backtrack in constant time, the authors proposed to use perfect hashing translating the encoded path to the node index.

6.6. Utilizing Ray Locality

Spatial data structures exploit the spatial locality of scene primitives. However, we can also use the spatial locality of rays to accelerate the traversal further. Ray reordering is a typical example of utilizing ray locality. Ray coherence is improved by grouping similar rays in order to increase cache hit ratios and control flow. This topic was recently surveyed by Meister et al. [MBGB20].

Hendrich et al. [HPMB19] proposed a ray classification scheme mapping rays to interior nodes deeper in the tree, skipping nodes in top levels. The idea is to uniformly subdivide ray space where each cell contains a short list of interior nodes that intersect a shaft corresponding to this cell. The space is first subdivided into a regular grid, and then each cell of this grid is further subdivided according to the directional component. Using this subdivision, a ray can be simply mapped into a particular cell and test only nodes associated with this cell. Similarly, Demoullin et al. [DGA19] proposed to use ray hashing to skip the interior nodes. The hash value is directly extracted from the floating-point representation of origin and direction. A caveat is that there is no guarantee that the found intersection is the nearest one. In general, it pays off to use these methods only if the overhead is lower than the speedup, which might be difficult if the classical ray traversal itself is already very fast.

It is also possible to efficiently traverse the BVH for a whole group of coherent rays. Instead of tracing rays sequentially, a bounding frustum containing the rays is traced through the BVH in the same manner as an individual ray. As the speedup of this approach heavily depends on the SIMD utilization, Benthin and Wald [BW09] proposed to trace several frusta (corresponding to the width of the SIMD unit) of coherent rays simultaneously and even generate the rays inside each frustum on the fly. In the case that the ray coherence is high enough, and therefore the extent of the ray frustum is sufficiently small, the necessary fallback of testing individual rays in case the frustum intersection tests fail, can be completely omitted.

Garanzha and Loop [GL10] extended this idea to ray tracing on GPUs. In the first step, rays are sorted using a multi-dimensional hashing. As successive rays often have the same hash value, the sorting time is reduced by first compacting the ray buffer based on the hash value. From each set of rays with an identical hash value, many ray packets are generated, such that the amount of rays per packet is below a given threshold. After calculating a frustum for each packet, a breadth-first traversal of the hierarchy per frustum is performed. While an 8-wide BVH with the sign heuristic is used for traversal, the approach is independent of the actual type of hierarchy. All leaf nodes visited during traversal are stored for each frustum. The final ray primitive intersections are then calculated in a separate kernel, where each thread handles a single ray. Again, the occupancy is maximized using the persistent threads approach.

6.7. Hardware Acceleration

Given that BVH traversal typically represents a substantial portion of modern ray tracing workloads, significant attention has been given to research efforts that aim to develop specialized hardware for BVH traversal and the associated task of triangle intersection. This body of research is substantially larger than the literature that exists on BVH construction hardware (Section 4.8). The research is varied and includes themes such as fixed-function designs for core operations, low-precision arithmetic, scheduling and data management to reduce memory bandwidth, and works that explore the merits of differing fundamental processor organizations (SIMD vs. MIMD). Many works in this area have focused purely on the core operations of BVH traversal and triangle intersection alone, while others have explored incorporating such units into more fully-featured architectures.

Fixed-Function Designs for BVH Traversal An early example of hardware BVH traversal features in the work of Fender and Rose [FR03], who prototyped their design on FPGAs. The design is dedicated to traversal and intersection operations. A BVH with three levels is used, and node boxes are specified as a set of arbitrary quadrilaterals. To traverse this format, the design re-purposes the intersection testing unit via additional control logic.

A later example of dedicated BVH traversal hardware appearing in the literature is the *T&I engine* [NPP*11]. Originally designed to process KD-trees, it was later improved and adapted for BVHs in the SGRT GPU [LSL*13]. The adapted unit consists of a *ray dispatch unit*, *traversal units* (TRV), and *intersection units* (IST). A major feature of this design is that each TRV possesses a *ray accumulation unit* (RAU). If a node access in the TRV misses the cache, rays are postponed in the RAU while the data is fetched. While processing the miss, further rays can be processed which may also be postponed as needed. Once a node fetch completes, all rays requiring the data can be processed. Other innovations include a three-stage IST design, which minimizes arithmetic unit complexity. The SGRT GPU featured an improved parallel design for the T&I engine which effectively allows rays to "branch" efficiently depending on whether the TRV requires a leaf node, internal node, or stack access operation (a short stack is used). This design was further extended to a parallel, Two-AABB intersection, which processes sibling nodes in parallel for more efficient pipelined processing [LLS*14]. Viitanen et al. [VKJT16] proposed an MBVH intersection architecture based on this unit. Lee et al. [LSH*15] proposed *reorder buffer* which represents an alternative to the RAU for latency hiding. While the RAU requires significant storage, reorder buffer instead allows rays to bypass stages of the traversal pipeline on cache misses, and follow a feedback loop back to the input buffer to re-attempt processing.

Davidovič et al. [DMS11] updated Woop's *ray traversal engine* (RTE) [Woo06] (originally devised for B-KD trees) for use with BVHs. The RTE is composed of two major units: a *traversal processor*, which traverses the acceleration structure, and a *geometry processor* which performs triangle intersections. The design also provides separate caches for node and geometry data, as well as an on-chip traversal stack. A major feature of the updated architecture is the adoption of the *treelet scheduling* technique of Aila and Karras [AK10]. Davidovič et al. test the integration of the design into a general-purpose GPU, and test the effect of both tail-recursive and continuation shaders on the performance of the RTE.

Over the last decade, the use of quantized BVHs and reduced-precision arithmetic for BVH traversal has become an important feature of the most efficient traversal implementations. This trend is evident in the hardware literature. Keely [Kee14] observed that traversal of quantized BVHs could occur in reduced-precision, allowing for large reductions in circuit area for hardware implementations. Keely's traversal scheme is based on the insight that by using quantized BVHs and by incrementally moving the ray's origin closer to the current node AABB on each traversal step, it is possible to ensure accurate results with fewer bits. Keely's work also introduces a compressed BVH node format, allowing for large reductions in node bandwidth. Building on this work, Vaidyanathan et al. [VAMS16] proposed an improved scheme for incremental, reduced-precision traversal and demonstrated that the algorithm guarantees watertightness while further reducing circuit area. The new scheme allows for an important optimization which is not possible in Keely's scheme, which is the reuse of intersection distances for shared parent/child bounding planes. Vaidyanathan et al. also proposed a novel compressed node format, which takes advantage of parent/child plane sharing to reduce node size further. This traversal algorithm was employed in the ray tracing architecture proposed by Liktor and Vaidyanathan [LV16]. Aside from reduced-precision methods, other arithmetic schemes with potential relevance to hardware have been proposed for ray tracing, including works that leverage fixed-point arithmetic [FR03, HK07], and a combination of floating and fixed-point [HLS*15].

Fully-Featured BVH Traversal Architectures Aila and Karras [AK10] propose a SIMD architecture for ray tracing, which is designed to reduce arithmetic and memory divergence for incoherent ray distributions. The design consists of a set of *processors*, which execute *warps* consisting of multiple threads, similar to conventional GPUs. Each thread manages one ray. The authors note that the design could accommodate fixed-function traversal units, but choose to focus on the contribution of reducing memory traffic. Each processor possesses an L1 cache, and all processors share an L2 cache. A key contribution of this work is that of *treelet scheduling*. The BVH is divided into *treelets* (small subtrees within the total BVH) which are set to the size of either the L1 or L2 cache. The architecture maintains a set of ray *queues*, with queues assigned to treelets at runtime. Rays begin tracing at the root treelet, and as rays cross treelet boundaries, they are placed in the ray queues to be processed later when their required treelet is present in the cache. The architecture thus attempts to maximize the number of rays which are processed each time a treelet is loaded on-chip, reducing memory bandwidth. To reduce stack traffic, the architecture maintains a stack-top cache on chip, while keeping the remainder in DRAM. A second key innovation of this work is the inclusion of *work compaction* logic which detects when the SIMD utilization of a given warp has fallen below an efficient level, at which point the unit terminates the warp and diverts the remaining active rays to the *launcher* which is responsible for warp creation.

Keely [Kee14] proposed a scheme for incorporating the reduced-precision traversal and node compression techniques of the same work into a GPU-based ray tracing architecture which leverages the existing GPU resources to a great extent. Modeled on an AMD Hawaii GPU [AMD13], the scheme delegates ray traversal to a reduced-precision *traversal unit* (TU), while ray-polygon intersection and shading are delegated to the SIMT units of the GPU. The design borrows the aforementioned treelet scheduling technique [AK10], but manages all ray queues on-chip to reduce bandwidth. Currently active rays are stored in the register files, and a *register transpose unit* allows for switching between layouts preferred by the TU and the SIMT cores as needed. The flow is controlled by both a *pipeline scheduler*, which controls the execution of each of the thread types (traversal, intersection and shading), and a *treelet scheduler*, which determines the loading of treelets into the cache.

A common theme in the hardware traversal literature is the use of MIMD processor organizations as opposed to traditional SIMD/SIMT GPU approaches. This argument is motivated by the observation that incoherent ray distributions cause SIMD divergence in SIMD/SIMT designs, reducing compute throughput. An early work based on the MIMD approach is the *TRaX* architecture [SBK*08, KSBP08, SKKB09, KSBD10, SKBD12]. The design is fully programmable, and consists of a large number of *thread multiprocessors* (TMs) which consist of a number of simple, multi-threaded *thread processors* (TPs). At any given time, TPs process a single thread, corresponding to one ray. Each TP possesses a register file and a set of frequently-used arithmetic operations. Expensive, lower-utilization arithmetic units are shared across TPs in a TM. The TPs in each TM share an L1 cache, and a set of L2 caches are shared between TMs. TRaX exhibits good performance relative to more dedicated hardware of the era, while being fully programmable. Vasiou et al. [VSM*18] present an analysis of render time and energy cost of TRaX.

Kopta et al. [KSS*13, KSS*15] proposed *STRaTA*. STRaTA is a MIMD architecture based on TRaX, but incorporates two major improvements. The first improvement is the integration of treelet scheduling [AK10]. However, rather than storing rays in DRAM, Kopta et al.'s architecture re-purposes a section of the L2 cache into a dedicated, on-chip ray memory, eliminating off-chip ray bandwidth. The second improvement is the introduction of dynamically-configurable special-purpose pipelines, which allow SW to establish HW connections between on-chip arithmetic units in order to implement core ray tracing operations more efficiently.

Shurko et al. [SGK*17] introduced the concept of *dual streaming* for hardware ray tracing. Dual streaming divides the scene into *scene segments* (based on BVH treelets) and a *ray stream*, which is split into multiple *queues* which are assigned to segments. When a scene segment is read, all rays intersecting the segment exhaust all valid traversal paths within the segment using a local stack. As the rays encounter new segments (treelet boundaries), rays are duplicated on the queues assigned to those segments. Thus, rays can only flow from parent to child segments, and each segment is streamed on-chip at most once per wavefront. A global set of hit records is used to merge results from multiple segments. An architecture is proposed for implementing dual streaming which includes the TM and TP organization of TRaX and STRaTA but also includes new features such as fixed-function pipelines for ray tracing operations, support for BVH treelets, a new *stream scheduler* for managing streams, and a *hit record updater*. Furthermore, the predictable, streaming memory access patterns are argued to be more favorable for modern DRAM systems, and the design exceeds the performance and energy efficiency of TraX and STRaTA.

The *Mach-RT* architecture [VSBY19, VSBY20] is a multi-chip design which builds upon the dual streaming approach to minimize ray traffic. The key insight of Mach-RT is that while it is desirable to store ray traffic on-chip, the resulting storage requirements are not feasible for a single chip. Mach-RT divides ray tracing across multiple *streaming processor chips* (SPCs), each processing separate regions of the image. The SPCs share an L3 cache. The design re-uses features of earlier works such as TM/TP organization, fixed-function pipelines and a dual streaming approach. However, the design includes new features such as an on-chip *wide vector buffer* for storing rays, an on-chip frame buffer, and an updated treelet scheduling mechanism designed to promote early ray termination. The design also removes the *hit record updater* of the dual streaming approach, as SPCs can update hit records locally since they process independent image regions. The design demonstrates improved rendering time and energy usage when compared to the STRaTA and dual streaming designs.

Liktor and Vaidyanathan [LV16] propose a MIMD ray tracing architecture employing node compression and reduced-precision traversal techniques. The fundamental building block of this design are the *traversal clusters*, which are further composed of *traversal units*, *leaf units* and a *primitive unit*. Traversal units are multi-threaded (one ray per thread) and are based on Vaidyanathan et al.'s algorithm [VAMS16]. Furthermore, they operate on a novel compressed node format and employ a short stack and restart trail technique [HSHH07, Lai10]. The leaf units fetch and process leaf nodes, but also special *glue nodes*, which reduce pointer sizes in the compressed BVH nodes. The leaf unit forwards requests to the primitive unit, which intersects primitives. Multiple traversal units and a single leaf unit form a *sub-cluster*, sharing an L1 cache. The sub-clusters share an L2 cache with the primitive unit, which shares its own L1 cache across multiple intersection pipelines. The design demonstrates that the use of compressed BVH nodes can greatly increase bandwidth efficiency, and permit more aggressive sharing of cache resources.

Lee et al. [LSL*13] proposed the *SGRT* Mobile Ray Tracing GPU architecture. SGRT is a heterogeneous mobile GPU which incorporates several key components to maximize performance and energy efficiency. The design is comprised of multiple *SGRT Cores*. Each SGRT core possesses a fixed-function T&I unit, and a *Samsung Reconfigurable Processor* (SRP) which is used for ray generation and shading. The SRPs are further composed of a *VLIW processor* and a *coarse-grained reconfigurable array* (CGRA), and are programmable with C language shaders. Coupled to the SGRT cores is a multi-core CPU, which is responsible for BVH build and update operations. The SGRT design was prototyped on an FPGA.

Nah et al. [NKP*15] introduce HART, a hybrid architecture for ray tracing animated scenes. The design consists of a CPU subsystem coupled with ray tracing hardware components. The CPU is responsible for building BVHs in an asynchronous fashion [IWP07] before sending them to the ray tracing hardware. The ray tracing hardware consists of two major units. The first unit is a T&I engine, based on earlier work [NPP*11], but optimized for BVHs rather than KD-trees. The second hardware unit is the *geometry and tree update unit* (GTU), which handles animation, computing primitive AABBs, computing triAccel structures [Wal04], and also features a hardware BVH refitting unit. The refitting unit updates the BVH while a full hierarchy is being prepared on the CPU. Shading is assumed to occur on conventional unified shader cores.

Other works of note include the SaarCOR [SWS02, SWW*04] and also the *RayCore* [NKK*14] architectures, which both represent ray tracing GPUs built around KD-trees. Kim et al. present a reconfigurable SIMT processor designed to address branch divergence [KKK10, KKK12]. Woop et al. [WSS05] presented the *RPU*, based on KD-trees, and the *DRPU* architecture [Woo06], based on B-KD trees [WMS06]. Finally, Deng et al. [DNL*17] also provide details on some of the work in this section.

Commercial Ray Traversal Hardware In addition to academic research, a small number of commercial ventures employing hardware ray tracing have appeared, with some based on BVH acceleration structures.

An early commercial venture was Advanced Rendering Technology (ART), which released custom ASIC designs and associated host systems for ray tracing [Gar18]. Two custom ASICs were released (the AR250 and AR350) and featured a custom CPU core, a *ray geometry engine*, a programmable *vector shading coprocessor* (based on RenderMan shaders), and an SDRAM interface. The ray geometry engine could intersect both triangles and parallelograms, and used the latter functionality for BVH traversal. The BVH itself was built on the host system without any custom hardware. The hardware was intended for high-end rendering applications, as exemplified by ART's *RenderDrive* host systems, which were based on a DEC Alpha system with custom PCI-X cards hosting a multitude of custom ASICs. Parallelism was achieved by distributing sets of rays across the custom chips, with geometry and BVH data being streamed to the chips on demand by the host system as traversal results are generated.

Imagination Technologies acquired Caustic Graphics [Des10] and released the *PowerVR Wizard GPU* [McC14, Bee20], which is primarily targeted for mobile applications. The hardware combines PowerVR's *unified shading clusters* (USCs) with several new units to support ray tracing. The *ray data master* is responsible for assembling ray shading tasks for the USCs. The *ray tracing unit* consists of an *intersection processor array*, for AABB and triangle intersection, and a *ray coherency engine*, which sorts rays for coherence. The *scene hierarchy generator* is capable of building BVHs in hardware (see Section 4.8). Finally, the *frame accumulator cache* supports frame buffer operations.

Modern ray tracing APIs such as Microsoft DXR [Mic20] and Vulkan Ray Tracing Extensions [Khr20] are fast becoming adopted in applications such as video games [DS19, SKAZ19]. NVIDIA RTX technology [NVI18], represents one of the most recent commercially available ray tracing hardware solutions, and was the first hardware to support these new APIs. A key hardware feature of RTX is the *RTCore*. RTCores are integrated into the Turing architecture's *streaming multiprocessors* (SM) and include hardware support for BVH ray traversal and triangle intersection. In addition to RTX, other major HW vendors have announced hardware support for ray tracing. Intel has announced that hardware ray tracing will feature in their upcoming Xe GPUs [Lil20]. AMD has announced the RDNA2 GPU Architecture, which features a *ray accelerator* (RA) [AMD20]. Also of note is that two next-generation consoles, the PlayStation 5 and the XBox Series X|S, employ the RDNA2 architecture, and have confirmed hardware ray tracing capability [Cer20, Tut20].

Other commercial work of note includes the *RayCore* GPU products from SiliconArts, which appear to be based on KD-trees [Sil20].

7. Rendering Frameworks

To our best knowledge, all modern production renderers use BVHs as their acceleration structures. Their implementations vary, and the differences arise from multiple factors, including whether they are commercial or non-commercial. In the case of in-house renderers, production workflows, pipelines, and each show's needs influence their design. The following descriptions are of the time when the papers were published, and because each renderer is in rapid development, there are likely deviations from the current implementation. However, the design decisions made by each development team reveal the trends.

Motion Blur One of the reasons for the widespread adoption of BVHs is that it is easier to implement motion blur when compared to other acceleration structures. Production renderers typically need to support motion blur with multiple motion segments. It is relatively easy to implement by linearly interpolating the primitives' vertices and the node bounds that enclose them, depending on the ray's time. SPI Arnold [KCSG18] builds a BVH that specializes in interpolating three frames: previous, present, and next. Object partitioning (i.e., not using spatial splits) makes motion blur implementation simple. In addition, a BVH built with object partitioning consumes less memory, works well for sufficiently tessellated objects, and requires performing an intersection test only once per primitive during BVH traversal.

Instancing Instancing is an indispensable feature for rendering large scenes. By creating copies of an object with different scales, positions, and shaders, users can generate, for example, sandy soil and forests with less memory. There are also differences in the implementations of instancing. Hyperion [BAC*18] and SPI Arnold [KCSG18] support single-level instancing while Autodesk Arnold [GIF*18] supports multi-level instancing, which allows users to handle even larger scenes. Since overlapping instanced objects lead to performance degradation, Hyperion [BAC*18] uses a method similar to partial re-braiding [BWWA17] taking into account the solid angle from a camera while generating references for the top-level hierarchy.

BVH Construction Autodesk Arnold [GIF*18] and Renderman [CFS*18] support on-demand construction. A BVH is built as a ray hits the object's bounding box to avoid construction costs for unused objects and to provide a quick response to users (i.e., to reduce the *time to first pixel*). In Autodesk Arnold, building starts when the first ray hits an object, and if other rays hit the same object during construction, those threads also participate in the building process. This parallelized construction is fast, scaling almost linearly with the number of processor cores. SPI Arnold [KCSG18] constructs BVHs for different objects in parallel before rendering starts. Manuka [FHL*18] adopts a unique shade-before-hit architecture, where all objects in a scene are first tessellated and shaded, and then a single BVH is built. The BVH built over a complete list of micropolygons makes intersection tests more efficient and eliminates the need for caching on-demand tessellated micropolygons.

Ray Tracing APIs The aim for a standardized ray tracing API dates back to the OpenRT API [DWBS03]. Due to lack of hardware support, interactive or real-time ray tracing has been limited to specialized hardware setups and specific applications. Later, Caustic Graphics [Des10] introduced the OpenRL API that provided an OpenGL-like interface to ray tracing hardware.

Instead of a stable API, the main hardware vendors released efficient *ray tracing kernels* such as Intel Embree [WWB*14] and NVIDIA OptiX [PBD*10]. These kernels were adopted by a number of commercial renderers as they offer a variety of ray tracing features, including dedicated hardware support. For instance, V-Ray [Cha20b] and Corona Renderer [Cha20a] use Embree, and V-Ray GPU and Autodesk Arnold [GIF*18] make use of OptiX.

In 2018, Microsoft announced the DirectX Ray Tracing API (DXR) [Mic20], which provides a standardized interface to ray tracing enabled graphics hardware. A platform-independent interface to ray tracing hardware is accessible through the Vulkan API extensions [Khr20]. Although DXR and Vulkan do not explicitly prescribe the use of a BVH for the acceleration data structure, the design of the API reflects the best practices established for BVHs by the research community, such as composing scenes with static and dynamic content using two-level hierarchies, and recomputing deformed objects by refitting. The actual data structure is opaque (i.e., hidden behind the API) and it can only be controlled indirectly by using appropriate build flags. NVIDIA was the first to provide DXR support with NVIDIA RTX hardware, while using BVHs for both the bottom and top-level data structures [NVI20]. NVIDIA provides an alternative ray tracing API through OptiX 7. This API goes beyond the strict two-level top-bottom acceleration data structure separation and provides generalized scene graph-like multi-level configurations [NVI20]. It is expected that other hardware vendors will release hardware with DXR / Vulkan support soon, most likely with BVHs being the core data structure for their implementations as well. For example, the PlayStation 5 and Xbox Series X next-gen game consoles are based on the new AMD RDNA2 architecture with DXR and Vulkan ray tracing support.

The standardization of real-time ray tracing APIs and the broad availability of ray tracing-enabled hardware has lead to the gradual adoption of the ray tracing paradigm by the game development community. Game engines now allow the possibility of using ray tracing mostly in the form of computing specific effects by ray tracing and computing the rest of the frame by using rasterization [HHM18, Ric20].

8. Other BVH Applications

This report focuses on ray tracing. However, BVHs can also be used for various other applications, such as illumination computation, direct volume rendering, and collision detection.

Illumination and Shadow Computation A cut formed on a BVH can be regarded as a rough approximation of the shape of the object, and it can be used to approximate occlusion calculations. Lacewell et al. [LBBS08] accelerated shadow computation by storing the directional opacity of the aggregate geometry in each node. Keul et al. [KKSM19] used a two-level BVH and stored visibility in a directional data structure at the bottom level to accelerate indirect illumination.

BVHs are a well-suited data structure for handling a large number of light sources. *Lightcuts* algorithms [WFA*05, WABG06, WBKP08, WKB12] aim to approximate illumination under a certain error bound by forming cuts. On the other hand, hierarchical importance sampling is typically used to obtain unbiased estimates [EK18, MC19, MPC19, Pan19, LXY19]. Estevez and Kulla [EK18] introduced a cost metric called the *surface area orientation heuristic* to cluster nearby and similarly oriented lights, allowing for efficient many-light sampling. Their algorithm can also be implemented on GPUs [MC19], and dynamic lighting can be handled with a two-level BVH [MPC19]. Pantaleoni [Pan19] took into account visibility by introducing fixed-size tree cuts. *BRDF-oriented light sampling* by Liu et al. [LXY19] considers the BRDF, light intensity, and the geometry term together. *Stochastic Lightcuts* [Yuk19] solved the shortcomings of Lightcuts such as sampling correlation by using stochastic sampling. Lin and Yuksel [LY20] further accelerated Stochastic Lightcuts with a complete binary tree. Ogaki [Oga20] built a BVH over negative space and applied it to portal sampling.

There are also examples of the use of BVHs in complex light transport algorithms. Fabianowski and Dingliana [FD09] showed that an LBVH constructed for photons is comparable in quality to a BVH constructed with the *voxel volume heuristic* [WGS04]. They also proposed an automatic method to determine the number of photons in leaf nodes. Otsu et al. [OHHD18] obtained an appropriate mutation step size for *Metropolis light transport* with the help of a BVH to prevent the decrease of the acceptance rate due to occlusion. Tokuyoshi and Harada [TH19] used a BVH to perform path connections efficiently in their *stochastic light culling framework*.

Volume Rendering Using BVHs for direct volume rendering is gradually gaining attraction because they allow for skipping empty space (i.e., fully transparent regions in sparse volume data), and can be seamlessly integrated into existing path tracing frameworks. Knoll et al. [KTW*11] processed regular grid volume data into bricks (each brick contains, for example, 4^3 voxels) and then constructed a BVH over non-empty bricks. By storing the minimum and maximum values in each node, empty space and constant subvolumes could be detected effectively during traversal. Performance was further improved by using coherent ray tracing. Similarly, Zellman et al. [ZHL19] built an LBVH over non-empty bricks. They quickly identified empty bricks via parallel voting. Instead of bricks, Ganter and Manzke [GM19] created clusters of non-empty voxels using a 3D version of a summed-area table to reduce BVH leaves and then built a BVH over those clusters. Knoll et al. [KWN*14] rendered volumes defined by particle data with *radial basis function* (RBF) kernels. The cost of the RBF field evaluation was again amortized by coherent ray tracing. This method can benefit from the construction algorithm developed for metaballs [GPB*09, GPP*10]. Morrical et al. [MUWP19] built a KD-tree over unstructured meshes and shrunk each cell to fit the primitives inside. They showed RTX-capable GPUs can accelerate the traversal because the resulting data structure is equivalent to a BVH built only with spatial splitting. Ströter et al. [SMSF20] proposed an octree-based LBVH suitable for volumetric meshes that fill space more densely than surface meshes. They used a set of Morton codes within the quantized AABB of each primitive and constructed an 8-ary BVH directly by advancing Morton codes by 3 bits.

Collision Detection Ytterlid and Shellshear [YS15] proposed a suitable metric for distance queries. Binary ostensibly-implicit trees [CDK20] are ideal for real-time collision detection because of their fast construction time. There is a large amount of literature on the use of BVHs in the context of collision detection. However, it is outside the scope of this paper. We refer the interested readers to the book by Ericson [Eri04] for collision detection-specific algorithms.

There is a small amount of work which explores sharing BVHs between both ray tracing and collision detection. Fowler et al. [FDM14] propose a method by which a single BVH can be efficiently shared between these two applications. The authors note that the desired leaf sizes often differ between ray tracing and collision detection, and introduce the concept of *inner leaves*. Components of a simulation preferring larger leaf sizes can effectively terminate traversal at larger, internal nodes, while others can pass through the inner leaves and terminate at the regular leaves as normal. This allows both components to make optimal use of the shared BVH.

Embree [WWB*14] also supports collision detection queries. The existing BVHs of two scenes are used to accelerate broad-phase collision detection (collision detection between objects). In contrast, the implementation of the narrow phase (collision detection between geometric primitives) must be implemented by the user through a programmable callback function.

Many other applications can benefit from BVHs, such as sound propagation [LCM07, CLT*08, Cer20] and N-body simulation [Ols18]. Due to space constraints, we limit ourselves to these examples.

9. Best Practices

In this report, we surveyed many interesting papers varying from basic algorithms to more complex ones. It might sometimes be difficult in any given case to choose the right algorithm, especially for those new in this field. Hence, in this section, we present a subset of algorithms that are relatively easy to implement while providing significant benefits in general use cases.

Construction For static geometry, we recommend starting with top-down construction using sweeping with the SAH-based cost function [GS87, MB90] as it is easy to implement and provides BVHs of relatively good quality. To speed up this process, one can use binning [WH06] instead of sweeping, which can be easily parallelized using multithreading and SIMD (see Section 4.1).

We also recommend considering the use of spatial splits [SFD09, PGDS09] during the top-down construction as they provide a significant performance gain, especially for scenes with long thin diagonal primitives, overlapping primitives, and primitives of non-uniform size (see Section 5.1). In such scenes spatial splits can easily provide 25% or more ray tracing speedup. On the other hand, in scenes with detailed, finely tessellated geometry, such as those used in movie production, the performance gain of using spatial splits can be marginal. Hence, most production renderers do not use spatial splits, preferring reliability and simplicity over performance. In any case, supporting spatial splits requires significant engineering effort, particularly for efficiently handling dynamic geometry updates or motion blur.

We can improve BVH quality by using optimization algorithms such as tree rotations [Ken08] or insertion-based optimization [BHH13], which are not difficult to implement (see Section 4.5). Subtree collapsing is another easy way to improve BVH quality by optimizing the sizes of leaves (see Section 4.6). One can consider using wide BVHs [WBB08, EG08], which might be beneficial, especially for incoherent rays. To construct a wide BVH, we can use a direct top-down construction [WBB08, WWB*14], or we can easily convert a binary BVH to a wide one using one of the collapsing algorithms [Pin10, YKL17] (see Section 5.2).

As we mentioned before, data layout may also have a significant impact on performance. Some algorithms, such as top-down construction, produce BVHs with relatively good data layout. However, some algorithms, such as LBVH [Kar12] or insertion-based optimization [BHH13], produce BVHs with a far from optimal data layout. Using simple DFS (on CPU) or BFS (on GPU) may improve the performance considerably in such cases (see Section 4.7).

Interactive Applications For interactive or real-time applications, a two-level hierarchy [WBS03] is easy to implement and provides many benefits such as instancing or rigid body animations. It can also be used to separate static and dynamic objects (see Section 5.3.3). For static objects, we can precompute high-quality BVHs offline using a complex algorithm such as insertion-based optimization [BHH13] as it will be reused many times. However, we have to deal with dynamic changes very quickly at runtime (see Section 4.5). Suppose the dynamic geometry is known a priori (or at least some representative positions). In that case, we recommend using T-SAH [BM15], which can be easily integrated into the insertion-based optimization method to optimize the BVH for the whole animation or the representative positions (see Section 5.3.1). If the changes are not known a priori or are too significant, it pays off to reconstruct the BVH from scratch using high-performance construction algorithms such as LBVH [Kar12, Ape14] or PLOC [MB18a], that are easy to implement, or the slightly more complex TRBVH [KA13, DP15]. LBVH is extremely fast, but it provides rather inferior BVHs as it is restricted to spatial median splits, which can be partially improved by incorporating the size of primitives into Morton codes [VBH17] (see Section 4.4). PLOC provides high-quality BVHs, but it is slower in comparison with LBVH (see Section 4.2). The choice depends on how many rays are traced and the complexity of the individual objects. Fast construction algorithms are also desirable if we want to insert new geometry into the scene at runtime since we do not want to wait too long for a response.

Traversal Traversal can be easily parallelized via SIMD, by testing multiple rays against one node/primitive. However, for incoherent rays, we recommend using wide BVHs with SIMD testing of one ray against multiple boxes/primitives [WBB08, EG08] (see Sections 5.2 and 6.1.1). On GPUs, implementing wide BVH traversal might be somewhat complicated, and the performance gain is not so significant. We thus recommend sticking with binary BVHs using a per-thread stack [AL09] (see Section 6.4). For complex scenes with finely tessellated geometry, we recommend the robust traversal by Ize [Ize13] to avoid numerical issues (see Section 6.2).

CPU versus GPU The platform used is a matter of choice which is heavily dependent on the target application. Both CPUs and GPUs have their advantages and disadvantages. Contemporary GPUs consist of thousands of cores providing enormous computational power. Thus, the GPU as a standard component of commodity computers is more suitable for interactive and real-time applications, especially with hardware acceleration. Limiting factors are smaller memory capacity and costly CPU-GPU transfers. Ideally, data are transferred to the GPU only once, while other computations take place entirely on the GPU. Designing GPU algorithms is also more challenging. On the other hand, the amount of the CPU memory could be an order of magnitude larger than the amount of the GPU memory, which is necessary if we have to deal with large complex scenes such as those in production. Besides thread parallelism, we can also relatively easily parallelize computations through multiple computers (e.g., HPC clusters), which might be useful, for example, for industrial visualizations where we are not limited by commodity hardware.

Available Implementations Many algorithms have publicly available implementations. Regarding CPU algorithms, we refer to Embree [WWB*14], which contains high-performance construction algorithms and traversal kernels. Nonetheless, it might not be easy for someone new in the field to read the code. In that case, we refer to PBRT [PJH16], which contains a simpler implementation of BVH construction and traversal. Regarding GPU algorithms, we refer to Aila's framework [AL09], which contains a stack-based traversal kernel using BVHs constructed by SBVH [SFD09]. This framework was later extended with other algorithms such as PLOC [MB18a], TRBVH with agglomerative clustering [DP15], and CPU-style traversal of wide BVHs [LSS18].

10. Open Problems

We have covered a multitude of studies so far, but there are still unresolved and unanswered questions. Research is driven by the ever-increasing complexity of data handled in games, movies, and scientific visualizations, and we expect to see the emergence of methods for handling it more efficiently.

Constructing SAH-optimal BVHs is believed to be NP-hard. Although the existing optimization algorithms produce high-quality BVHs, we do not know how far we are from the optimum. Whether we can do better than the existing techniques is an open question. It would be interesting to show more theoretical properties such as the NP-hardness of the problem, or propose an approximation algorithm that bounds the distance from the optimum. Another problem is that there is a gap between the SAH cost and the actual ray tracing performance. The EPO is a more accurate metric, but it is not as easy to calculate during the construction phase. Another problem with building BVHs is that it is very memory bandwidth-intensive, which is the primary bottleneck for reducing run-time costs. A research focus should be on algorithms that reduce memory bandwidth during construction. Top-down approaches are less bandwidth-hungry than bottom-up approaches due to caches.

Wide BVHs are increasingly popular as they are suitable for parallel processing. Typically, we build wide BVHs directly in a top-down manner using multiple splits to fill all slots in the node. Another option is to convert a binary BVH to a wide BVH in an optimal way using collapsing algorithms [Pin10, YKL17]. The question is whether we can get a better BVH by directly optimizing a wide BVH instead of converting an optimized binary BVH to a wide one. Nonetheless, it is often not obvious how to extend more complex algorithms, such as insertion-based optimization [BHH13], to directly process wide BVHs. A naïve extension would result in moving nodes to empty slots in the upper levels and removing unnecessary nodes in bottom levels completely from the tree, which is equivalent to the greedy collapse. The problem is that after the collapse the optimization gets stuck in a local minimum that is almost impossible to get out of. To prevent this, we could perform optimization on a binary BVH while using a dynamic programming-based approach to evaluate the cost function of the resulting BVH after the collapse. In this case, the cost function is piece-wise constant. A topological change in the binary BVH may result in no change in the cost function, for example, when we modify nodes that will be discarded after the collapse. Furthermore, it is unclear how to efficiently update the dynamic programming information when the topological change is made.

Ray distribution based optimization methods [BH09, FLF12, WSWG13, GHB15, ODJ16] suffer from memory overhead due to the variables used to collect statistics such as the number of node accesses, performance degradation caused by atomic operations, and an additional preprocessing pass. Additionally, representing statistics for a given BVH is generally not sufficient for constructing new optimized ones. Successfully addressing these problems would enable various scene-specific optimizations.

In most cases, a BVH is traversed from the root node. Dammertz et al. [DHK08] stored references to the parent and traversed the ray from intermediate nodes instead of from the root to speed up shadow computation. Knowing the parent node's position or quickly identifying the lowest common ancestor of two leaves helps to reduce traversal steps. It can be easily done with a complete tree. However, usually in SAH-optimized BVHs, the parent pointer is not stored due to memory and alignment constraints, and thus the use of such techniques is limited. Wald et al.'s [WAB18] iterative traversal reuses the previous state. Storing traversal states of a large batch of rays can be problematic in terms of memory consumption.

The increase in data complexity is not only because of the increasing amount of data, but also because the data takes various forms. Unsurprisingly, most of the literature focuses on polygonal objects (mostly triangles), and there is a limited amount of work dealing with non-polygonal objects such as curves and isosurfaces. In particular, hair and fur rendering are crucial for human and animal characters, and in recent years, fabrics are rendered at the yarn level to improve realism. Even with state of the art methods, building a BVH for non-polygonal objects is time-consuming, and computing tight bounding boxes is not always easy. Various primitives, including points, curves, and triangles, are used, particularly in production rendering. How to handle a mixture of different primitives (with and without motion blur) is not well understood. There are currently two possible options: One is to build a BVH for all primitive types; and the other is to build a bottom-level acceleration structure, one for each primitive type, and then build a top-level acceleration structure over them. The previous work by Han et al. [HWU*19] has a small discussion on this topic. In addition, the optimal data structure varies with changes in the production pipeline. For instance, when formerly separate processes are integrated (e.g., physics simulation and rendering), a single data structure serving multiple purposes is desired. Fowler et al. [FDM14] introduced a shared BVH that can be used for both ray tracing and collision detection. In recent versions of Embree [WWB*14], the same BVH can also be used for collision detection. As discussed in Section 8, the number of applications that utilize BVHs continues to grow; therefore, a universal BVH should satisfy requirements arising from various perspectives, including memory efficiency, data structure update overhead, and development and maintenance burden.

11. Conclusion

As of today, BVHs stand at the core of most ray tracing frameworks. The current situation is a result of decades of research on spatial data structures for ray tracing. In this relatively long time span, there was no survey specifically targeting BVHs, despite a great deal of research effort devoted to BVHs in the last 15 years.

This report surveyed core concepts behind BVHs for ray tracing, including the SAH cost model and its extensions. We discussed the existing BVH construction algorithms ranging from the widely used top-down methods to specialized optimization-based techniques. We surveyed low level concepts such as efficient data layout and hardware acceleration of BVH construction. The report also covers various extensions of BVHs including spatial splits, wide BVHs, dynamic scene support, non-polygonal objects, and compact representations.

We surveyed existing BVH traversal algorithms by discussing different traversal orders, numerical issues, stream traversal, GPU and stackless traversal, exploiting ray locality, and hardware-accelerated ray traversal. We provided a brief overview of the state of the art rendering frameworks and APIs using BVHs. We listed other applications besides ray tracing that exploit BVHs. Finally, we concluded the survey by summarizing best practice recommendations and listing open problems related to BVHs.

Acknowledgements

We would like to thank Toshiya Hachisuka for thoughtful feedback and comments. This work was supported by the Czech Science Foundation (GA18-20374S) and Research Center for Informatics (CZ.02.1.01/0.0/0.0/16_019/0000765).

References

[Áfr13] ÁFRA A. T.: *Faster Incoherent Ray Traversal Using 8-Wide AVX Instructions*. Tech. rep., 2013. 14

[AGGW15] AMSTUTZ J., GRIBBLE C., GNTHER J., WALD I.: An Evaluation of Multi-Hit Ray Traversal in a BVH using Existing First-Hit/Any-Hit Kernels. *Journal of Computer Graphics Techniques 4*, 4 (2015), 72–88. 15

[AK10] AILA T., KARRAS T.: Architecture Considerations for Tracing Incoherent Rays. In *Proceedings of High-Performance Graphics* (2010), p. 113–122. 18, 19

[AKL13] AILA T., KARRAS T., LAINE S.: On Quality Metrics of Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics* (2013), pp. 101–108. 4, 5, 10

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High-Performance Graphics* (2009), pp. 145–149. 16, 17, 23

[AMD13] ADVANCED MICRO DEVICES I.: Sea Islands Instruction Set Architecture, 2013. Online Accessed: November 2nd 2020. URL: http://developer.amd.com/wordpress/media/2013/07/AMD_Sea_Islands_Instruction_Set_Architecture.pdf. 19

[AMD20] ADVANCED MICRO DEVICES I.: RDNA2 Architecture, 2020. Online Accessed: November 2nd 2020. URL: https://www.amd.com/en/technologies/rdna-2. 20

[Ape14] APETREI C.: Fast and Simple Agglomerative LBVH Construction. In *Proceedings of Computer Graphics and Visual Computing* (2014). 7, 23

[App68] APPEL A.: Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of Spring Joint Computer Conference* (1968), pp. 37–45. 1

[ASK14] ÁFRA A., SZIRMAY-KALOS L.: Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing. *Computer Graphics Forum 33*, 1 (2014), 129–140. 17

[BAC*18] BURLEY B., ADLER D., CHIANG M. J.-Y., DRISKILL H., HABEL R., KELLY P., KUTZ P., LI Y. K., TEECE D.: The Design and Evolution of Disney's Hyperion Renderer. *ACM Transaction on Graphics 37*, 3 (2018), 33:1–33:22. 21

[BAM13] BARRINGER R., AKENINE-MÖLLER T.: Dynamic Stackless Binary Tree Traversal. *Journal of Graphics Tools 2*, 1 (2013), 38–49. 17

[BAM14] BARRINGER R., AKENINE-MÖLLER T.: Dynamic Ray Stream Traversal. *ACM Transactions on Graphics 33*, 4 (2014), 151:1–151:9. 14, 16

[Bee20] BEETS K.: What's Your Level? The Six Levels of Ray Tracing Acceleration. Imagination Technologies White Paper, 2020. 20

[BEM10] BAUSZAT P., EISEMANN M., MAGNOR M.: The Minimal Bounding Volume Hierarchy. In *Proceedings of Vision, Modeling, and Visualization* (2010). 13

[BH09] BITTNER J., HAVRAN V.: RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In *Proceedings of Spring Conference on Computer Graphics* (2009), pp. 61–67. 4, 15, 24

[BHH13] BITTNER J., HAPALA M., HAVRAN V.: Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum 32*, 1 (2013), 85–100. 6, 7, 8, 11, 22, 23

[BHH15] BITTNER J., HAPALA M., HAVRAN V.: Incremental BVH Construction for Ray Tracing. *Computers and Graphics 47*, 1 (2015), 135–144. 6

[BK16] BINDER N., KELLER A.: Efficient Stackless Hierarchy Traversal on GPUs with Backtracking in Constant Time. In *Proceedings of High-Performance Graphics* (2016), pp. 41–50. 17

[BM15] BITTNER J., MEISTER D.: T-SAH: Animation Optimized Bounding Volume Hierarchies. *Computer Graphics Forum (Proceedings of Eurographics) 34*, 2 (2015), 527–536. 7, 11, 23

[BW09] BENTHIN C., WALD I.: Efficient Ray Traced Soft Shadows Using Multi-Frusta Tracing. In *Proceedings of the Conference on High-Performance Graphics* (2009), p. 135–144. 18

[BWWA17] BENTHIN C., WOOP S., WALD I., ÁFRA A.: Improved Two-Level BVHs using Partial Re-Braiding. In *Proceedings of High-Performance Graphics* (2017). 11, 21

[BWWA18] BENTHIN C., WALD I., WOOP S., ÁFRA A. T.: Compressed-leaf Bounding Volume Hierarchies. In *Proceedings of the Conference on High-Performance Graphics* (2018), pp. 6:1–6:4. 12

[CDK20] CHITALU F. M., DUBACH C., KOMURA T.: Binary Ostensibly-Implicit Trees for Fast Collision Detection. *Computer Graphics Forum 39*, 2 (2020), 509–521. 7, 14, 22

[Cer20] CERNY M.: The Road to PS5. Sony Playstation Online Talk, 2020. Online Accessed: November 2nd 2020. URL: https://www.youtube.com/watch?v=ph8LyNIT9sg. 20, 22

[CFLB06] CHRISTENSEN P., FONG J., LAUR D., BATALI D.: Ray Tracing for the Movie 'Cars'. *Proceedings of IEEE Symposium on Interactive Ray Tracing* (2006), 1–6. 11

[CFS*18] CHRISTENSEN P., FONG J., SHADE J., WOOTEN W., SCHUBERT B., KENSLER A., FRIEDMAN S., KILPATRICK C., RAMSHAW C., BANNISTER M., RAYNER B., BROUILLAT J., LIANI M.: RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. *ACM Transactions on Graphics 37*, 3 (2018), 30:1–30:21. 21

[Cha20a] CHAOS CZECH A.S.: Corona Renderer, 2020. URL: https://corona-renderer.com/features/proudly-cpu-based. 21

[Cha20b] CHAOS SOFTWARE LTD.: V-Ray for 3ds Max Help, 2020. URL: https://docs.chaosgroup.com/display/VMAX/System. 21

[Cla76] CLARK J.: Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM 19*, 10 (1976), 547–554. 2

[CLT*08] CHANDAK A., LAUTERBACH C., TAYLOR M., REN Z., MANOCHA D.: AD-Frustum: Adaptive Frustum Tracing for Interactive Sound Propagation. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1707–1722. 22

[CSE06] CLINE D., STEELE K., EGBERT P.: Lightweight Bounding Volumes for Ray Tracing. *Journal of Graphics Tools 11*, 4 (2006), 61–71. 13

[DBBS06] DUTRE P., BALA K., BEKAERT P., SHIRLEY P.: *Advanced Global Illumination*. AK Peters Ltd, 2006. 1

[Des10] DESIGN&REUSE: Imagination to Acquire Caustic Graphics, Developer of Real-Time Ray-Tracing Graphics Technology, for $27 Million, 2010. Online Accessed: November 2nd 2020. URL: https://www.design-reuse.com/news/25160/imagination-caustic-acquisition.html. 20, 21

[DFM12] DOYLE M., FOWLER C., MANZKE M.: Hardware Accelerated Construction of SAH-based Bounding Volume Hierarchies for Interactive Ray Tracing. In *Proceedings of Symposium on Interactive 3D Graphics and Games* (2012), pp. 209–209. 8

[DFM13] DOYLE M., FOWLER C., MANZKE M.: A Hardware Unit for Fast SAH-optimised BVH Construction. *ACM Transactions on Graphics 32*, 4 (2013), 139:1–139:10. 8

[DGA19] DEMOULLIN F., GUBRAN A. A., AAMODT T. M.: Hash-Based Ray Path Prediction: Skipping BVH Traversal Computation by Exploiting Ray Locality. *arXiv e-prints* (2019), arXiv:1910.01304. 17

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum 27* (2008), 1225–1233(9). 10, 14, 24

[DK08] DAMMERTZ H., KELLER A.: Edge Volume Heuristic - Robust Triangle Subdivision for Improved BVH Performance. In *Proceedings of Symposium on Interactive Ray Tracing* (2008), pp. 155–158. 9, 10

[dLPP14] D. L. PAHINS C. A., POZZER C. T.: Improving Divide-and-Conquer Ray-Tracing Using a Parallel Approach. In *2014 27th SIBGRAPI Conference on Graphics, Patterns and Images* (2014), pp. 9–16. 13

[DMS11] DAVIDOVIC T., MARSALEK L., SLUSALLEK P.: Performance Considerations When Using a Dedicated Ray Traversal Engine. In *Proceedings of International Conference on Computer Graphics, Visualization and Computer Vision* (2011). 18

[DNL*17] DENG Y., NI Y., LI Z., MU S., ZHANG W.: Toward Real-Time Ray Tracing : A Survey on Hardware Acceleration and Microarchitecture Techniques. *ACM Computing Surveys 50*, 4 (2017), 58:1–58 : 41. 9, 20

[DP15] DOMINGUES L., PEDRINI H.: Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring. In *Proceedings of High-Performance Graphics* (2015), pp. 13–20. 7, 23

[DS19] DELIGIANNIS J., SCHMID J.: It Just Works: Ray-Traced Reflections in Battlefield V. In *Game Developer's Conference (GDC) 2019* (2019). 20

[DTM17] DOYLE M. J., TUOHY C., MANZKE M.: Evaluation of a BVH Construction Accelerator Architecture for High-Quality Visualization. *IEEE Transactions on Multi-Scale Computing Systems PP* (2017), 1–1. 8

[DWBS03] DIETRICH A., WALD I., BENTHIN C., SLUSALLEK P.: The OpenRT Application Programming Interface–Towards a Common API for Interactive Ray Tracing. In *Proceedings of OpenSG Symposium* (2003), pp. 23–31. 21

[EBM12] EISEMANN M., BAUSZAT P., MAGNOR M.: *Implicit Object Space Partitioning: The No-Memory BVH*. Tech. Rep. 16, Computer Graphics Lab, TU Braunschweig, 2012. 13

[EG07] Ernst M., Greiner G.: Early Split Clipping for Bounding Volume Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing* (2007), pp. 73–78. 9, 10

[EG08] Ernst M., Greiner G.: Multi Bounding Volume Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing* (2008), pp. 35–40. 10, 14, 22, 23

[EK18] Estevez A. C., Kulla C.: Importance Sampling of Many Lights with Adaptive Tree Splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (2018), 25:1–25:17. 21

[Eri04] Ericson C.: *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., 2004. 22

[ESSL10] Enderton E., Sintorn E., Shirley P., Luebke D.: Stochastic Transparency. In *Proceedings of Symposium on Interactive 3D Graphics and Games* (2010), pp. 157–164. 12

[ESV96] Evans F., Skiena S., Varshney A.: Optimizing Triangle Strips for Fast Rendering. In *Proceedings of Visualization* (1996), pp. 319 – 326. 13

[EW11] Ernst M., Woop S.: Ray Tracing with Shared-Plane Bounding Volume Hierarchies. *Journal of Graphics, GPU, and Game Tools 15*, 3 (2011), 141–151. 13

[EWM08] Eisemann M., Woizischke C., Magnor M.: Ray Tracing with the Single Slab Hierarchy. In *Proceedings of Vision, Modeling, and Visualization* (2008), pp. 373–381. 13

[FD09] Fabianowski B., Dingliana J.: Interactive Global Photon Mapping. In *Proceedings of the Eurographics Symposium on Rendering* (2009), p. 1151–1159. 13, 22

[FDM14] Fowler C., Doyle M. J., Manzke M.: Adaptive BVH: An Evaluation of an Efficient Shared Data Structure for Interactive Simulation. In *Proceedings of Spring Conference on Computer Graphics* (2014), p. 37–45. 22, 24

[FFD09] Fabianowski B., Fowler C., Dingliana J.: A Cost Metric for Scene-Interior Ray Origins. In *Proceedings of Eurographics (Short Papers)* (2009). 4

[FHL*18] Fascione L., Hanika J., Leone M., Droske M., Schwarzhaupt J., Davidovič T., Weidlich A., Meng J.: Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production. *ACM Transactions on Graphics 37*, 3 (2018), 31:1–31:18. 21

[FLF12] Feltman N., Lee M., Fatahalian K.: SRDH: Specializing BVH Construction and Traversal Order Using Representative Shadow Ray Sets. In *Proceedings of High-Performance Graphics* (2012), pp. 49–55. 15, 24

[FLP*17] Fuetterling V., Lojewski C., Pfreundt F.-J., Hamann B., Ebert A.: Accelerated Single Ray Tracing for Wide Vector Units. In *Proceedings of High-Performance Graphics* (2017). 14

[FLPE15] Fuetterling V., Lojewski C., Pfreundt F.-J., Ebert A.: Efficient Ray Tracing Kernels for Modern CPU Architectures. *Journal of Computer Graphics Techniques 4*, 5 (2015), 90–111. 10, 14, 16

[FLPE16] Fuetterling V., Lojewski C., Pfreundt F.-J., Ebert A.: Parallel Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2016), pp. 21–30. 10

[FR03] Fender J., Rose J.: A High-Speed Ray Tracing Engine Built on a Field-Programmable System. In *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)* (2003), pp. 188–195. 18, 19

[Gar18] Garrard A.: Cold Chips: ART's RenderDrive Architecture – Ray Tracing Hardware from before the GPU. High Performance Graphics 2018 Hot3D Session, 2018. 20

[Gas16] Gasparian T.: *Fast Divergent Ray Traversal by Batching Rays in a BVH*. Master's thesis, Utrecht University, 2016. 16

[GBDAM15] Ganestam P., Barringer R., Doggett M., Akenine-Möller T.: Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees. *Journal of Computer Graphics Techniques 4*, 3 (2015), 23–42. 5, 8

[GD16] Ganestam P., Doggett M.: SAH Guided Spatial Split Partitioning for Fast BVH Construction. *Computer Graphics Forum* (2016). 10

[GHB15] Gu Y., He Y., Blelloch G. E.: Ray Specialized Contraction on Bounding Volume Hierarchies. *Computer Graphics Forum* (2015). 10, 24

[GHFB13] Gu Y., He Y., Fatahalian K., Blelloch G.: Efficient BVH Construction via Approximate Agglomerative Clustering. In *Proceedings of High-Performance Graphics* (2013), pp. 81–88. 6, 8

[GIF*18] Georgiev I., Ize T., Farnsworth M., Montoya-Vozmediano R., King A., Lommel B. V., Jimenez A., Anson O., Ogaki S., Johnston E., Herubel A., Russell D., Servant F., Fajardo M.: Arnold: A Brute-Force Production Path Tracer. *ACM Transactions on Graphics 37*, 3 (2018), 32:1–32:12. 12, 21

[GL10] Garanzha K., Loop C.: Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum 29*, 2 (2010), 289–298. 18

[GM19] Ganter D., Manzke M.: An Analysis of Region Clustered BVH Volume Rendering on GPU. *Computer Graphics Forum* (2019). 22

[GNK14] Gribble C., Naveros A., Kerzner E.: Multi-Hit Ray Traversal. *Journal of Computer Graphics Techniques 3*, 1 (2014), 1–17. 15

[GPB*09] Gourmel O., Pajot A., Barthe L., Paulin M., Poulin P.: BVH for Efficient Raytracing of Dynamic Metaballs on GPU. In *Proceedings of SIGGRAPH (Talks)* (2009). 12, 22

[GPBG11] Garanzha K., Premože S., Bely A., Galaktionov V.: Grid-Based SAH BVH Construction on a GPU. *The Visual Computer 27*, 6–8 (2011), 697–706. 5

[GPM11] Garanzha K., Pantaleoni J., McAllister D.: Simpler and Faster HLBVH with Work Queues. In *Proceedings of High-Performance Graphics* (2011), pp. 59–64. 7

[GPP*10] Gourmel O., Pajot A., Paulin M., Barthe L., Poulin P.: Fitted BVH for Fast Raytracing of Metaballs. *Computer Graphics Forum* (2010). 12, 22

[GR08] Gribble C., Ramani K.: Coherent Ray Tracing via Stream Filtering. In *Proceeding of Symposium on Interactive Ray Tracing* (2008), pp. 59–66. 16

[Gri16] Gribble C.: Node Culling Multi-Hit BVH Traversal. In *Proceedings of the Eurographics Symposium on Rendering (Experimental Ideas & Implementations)* (2016), pp. 85–90. 15

[Gri19] Gribble C.: *Multi-Hit Ray Tracing in DXR*. 2019, pp. 111–125. 15

[GS87] Goldsmith J., Salmon J.: Automatic Creation of Object Hierarchies for Ray Tracing. *Computer Graphics and Applications 7*, 5 (1987), 14–20. 3, 6, 22

[GSNK11] Gruenschloss L., Stich M., Nawaz S., Keller A.: MSBVH: An Efficient Acceleration Data Structure for Ray Traced Motion Blur. In *Proceedings of High-Performance Graphics* (2011). 11

[Gut14] Guthe M.: Latency Considerations of Depth-first GPU Ray Tracing. In *Proceedings of Eurographics (Short Papers)* (2014). 16, 17

[GWA16] Gribble C., Wald I., Amstutz J.: Implementing Node Culling Multi-Hit BVH Traversal in Embree. *Journal of Computer Graphics Techniques 5*, 4 (2016), 1–7. 15

[Hac15] Hachisuka T.: Implementing a Photorealistic Rendering System using GLSL. *arXiv e-prints* (2015), arXiv:1505.06022. 17

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000. 2, 10

[HDW*13] HAPALA M., DAVIDOVIČ T., WALD I., HAVRAN V., SLUSALLEK P.: Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings of Spring Conference on Computer Graphics* (2013), pp. 7–12. 17

[HHM18] HEITZ E., HILL S., MCGUIRE M.: Combining Analytic Direct Illumination and Stochastic. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2018* (2018). 21

[HHS06] HAVRAN V., HERZOG R., SEIDEL H.-P.: On the Fast Construction of Spatial Data Structures for Ray Tracing. In *Proceedings of Symposium on Interactive Ray Tracing* (2006), pp. 71–80. 10, 13

[HK07] HANIKA J., KELLER A.: Towards Hardware Ray Tracing using Fixed Point Arithmetic. In *Proceedings of Symposium on Interactive Ray Tracing* (2007), pp. 119–128. 19

[HKL10] HANIKA J., KELLER A., LENSCH H.: Two-Level Ray Tracing with Reordering for Highly Complex Scenes. In *Proceedings of Graphics Interface* (2010), pp. 145–152. 11

[HLS*15] HWANG S. J., LEE J., SHIN Y., LEE W.-J., RYU S.: A Mobile Ray Tracing Engine with Hybrid Number Representations. In *Proceedings of SIGGRAPH Asia (Mobile Graphics and Interactive Applications)* (2015). 19

[HMB17] HENDRICH J., MEISTER D., BITTNER J.: Parallel BVH Construction Using Progressive Hierarchical Refinement. *Computer Graphics Forum (Proceedings of Eurographics) 36*, 2 (2017), 487–494. 5, 8, 10

[HMF07] HUNT W., MARK W., FUSSELL D.: Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing* (2007), pp. 47–54. 5

[HPMB19] HENDRICH J., POSPÍŠIL A., MEISTER D., BITTNER J.: Ray Classification for Accelerated BVH Traversal. *Computer Graphics Forum 38*, 4 (2019), 49–56. 17

[HQL*10] HOU Q., QIN H., LI W., GUO B., ZHOU K.: Micropolygon Ray Tracing with Defocus and Motion Blur. *ACM Transactions on Graphics 29*, 4 (2010). 11

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive K-d Tree GPU Raytracing. p. 167–174. 20

[HWU*19] HAN M., WALD I., USHER W., WU Q., WANG F., PASCUCCI V., HANSEN C. D., JOHNSON C. R.: Ray Tracing Generalized Tube Primitives : Method and Applications. *Computer Graphics Forum* (2019). 24

[IH11] IZE T., HANSEN C. D.: RTSAH Traversal Order for Occlusion Rays. *Computer Graphics Forum 30* (2011), 297–305. 14

[IWP07] IZE T., WALD I., PARKER S.: Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of Symposium on Parallel Graphics and Visualization* (2007), pp. 101–108. 11, 20

[Ize13] IZE T.: Robust BVH Ray Traversal. *Journal of Computer Graphics Techniques 2*, 2 (2013), 12–27. 15, 23

[Jen01] JENSEN H. W.: *Realistic Image Synthesis Using Photon Mapping*. 2001. 13

[KA13] KARRAS T., AILA T.: Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics* (2013), pp. 89–100. 3, 7, 8, 9, 10, 23

[Kaj86] KAJIYA J.: The Rendering Equation. *SIGGRAPH Computer Graphics 20*, 4 (1986), 143–150. 2

[Kar07] KARRENBERG R.: *Memory Aware Realtime Ray Tracing: The Bounding Plane Hierarchy*. Master's thesis, Saarland University, 2007. 13

[Kar12] KARRAS T.: Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Proceedings of High-Performance Graphics* (2012), pp. 33–37. 7, 22, 23

[KCSG18] KULLA C., CONTY A., STEIN C., GRITZ L.: Sony Pictures Imageworks Arnold. *ACM Transactions on Graphics 37*, 3 (2018), 29:1–29:18. 21

[Kee14] KEELY S.: Reduced Precision for Hardware Ray Tracing in GPUs. In *Proceedings of High-Performance Graphics* (2014), p. 29–40. 18, 19

[Ken08] KENSLER A.: Tree Rotations for Improving Bounding Volume Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing* (2008), pp. 73–76. 7, 22

[Khr20] KHRONOS GROUP: Vulkan Ray Tracing Extensions Specification, 2020. URL: `https://www.khronos.org/blog/vulkan-ray-tracing-final-specification-release`. 20, 21

[KIS*12] KOPTA D., IZE T., SPJUT J., BRUNVAND E., DAVIS A., KENSLER A.: Fast, Effective BVH Updates for Animated Scenes. In *Proceedings of Symposium on Interactive 3D Graphics and Games* (2012), pp. 197–204. 11

[KKK10] KIM H.-Y., KIM Y.-J., KIM L.-S.: Reconfigurable Mobile Stream Processor for Ray Tracing. In *Proceedings of Custom Integrated Circuits Conference* (2010), pp. 1–4. 20

[KKK12] KIM H., KIM Y., KIM L.: MRTP: Mobile Ray Tracing Processor with Reconfigurable Stream Multi-Processors for High Datapath Utilization. *IEEE Journal of Solid-State Circuits 47*, 2 (2012), 518–535. 20

[KKSM19] KEUL K., KOSS T., SCHRÖDER F., MÜLLER S.: Combining Two-level Data Structures and Line Space Precomputations to Accelerate Indirect Illumination. pp. 228–235. 21

[KMKY10] KIM T.-J., MOON B., KIM D., YOON S.-E.: RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (2010), 273–286. 12

[KSBD10] KOPTA D., SPJUT J., BRUNVAND E., DAVIS A.: Efficient MIMD Architectures for High-Performance Ray Tracing. In *Proceedings of Computer Design* (2010), p. 9–16. 19

[KSBP08] KOPTA D., SPJUT J., BRUNVAND E., PARKER S.: Comparing Incoherent Ray Performance of TRaX vs. Manta. In *Proceedings of Symposium on Interactive Ray Tracing* (2008), pp. 183–183. 19

[KSS*13] KOPTA D., SHKURKO K., SPJUT J., BRUNVAND E., DAVIS A.: An Energy and Bandwidth Efficient Ray Tracing Architecture. In *Proceedings of High-Performance Graphics* (2013), p. 121–128. 19

[KSS*15] KOPTA D., SHKURKO K., SPJUT J., BRUNVAND E., DAVIS A.: Memory Considerations for Low Energy Ray Tracing. *Computer Graphics Forum 34*, 1 (2015), 47–59. 19

[KTW*11] KNOLL A., THELEN S., WALD I., HANSEN C. D., HAGEN H., PAPKA M. E.: Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal. In *Proceedings of Pacific Visualization Symposium* (2011), pp. 3–10. 22

[KWN*14] KNOLL A., WALD I., NAVRATIL P., BOWEN A., REDA K., PAPKA M. E., GAITHER K.: RBF Volume Ray Casting on Multicore and Manycore CPUs. In *Proceedings of the Eurographics Conference on Visualization* (2014), pp. 71–80. 22

[Lai10] LAINE S.: Restart Trail for Stackless BVH Traversal. In *Proceedings of High-Performance Graphics* (2010), pp. 107–111. 17, 20

[LBBS08] LACEWELL D., BURLEY B., BOULOS S., SHIRLEY P.: Raytracing Prefiltered Occlusion for Aggregate Geometry. In *Proceedings of Symposium on Interactive Ray Tracing* (2008), pp. 19–26. 21

[LCM07] LAUTERBACH C., CHANDAK A., MANOCHA D.: Interactive Sound Rendering in Complex and Dynamic Scenes using Frustum Tracing. *IEEE Transactions on Visualization & Computer Graphics 13*, 06 (2007), 1672–1679. 22

[LDNL15] LIU X., DENG Y., NI Y., LI Z.: FastTree: A Hardware KD-Tree Construction Acceleration Engine for Real-Time Ray Tracing. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition* (2015), p. 1595–1598. 9

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384. 5, 6

[Lil20] LILLY P.: A Closer Look At Intel's Xe-HPG GPU For Gamers With Ray Tracing, VRS And Image Sharpening, 2020. Online Accessed: November 2nd 2020. URL: https://hothardware.com/news/closer-look-intels-xe-hpg-gpu-gamers. 20

[LK11] LAINE S., KARRAS T.: Stratified Sampling for Stochastic Transparency. *Computer Graphics Forum* (2011). 12

[LL20] LEE W.-J., LIKTOR G.: Lazy Build of Acceleration Structures with Traversal Shaders. In *Proceedings of ACM SIGGRAPH Asia (Technical Communication)* (2020). 12

[LLS*14] LEE J., LEE W.-J., SHIN Y., HWANG S., RYU S., KIM J.: Two-AABB Traversal for Mobile Real-Time Ray Tracing. 18

[LSH*15] LEE W.-J., SHIN Y., HWANG S. J., KANG S., YOO J.-J., RYU S.: Reorder Buffer: An Energy-efficient Multithreading Architecture for Hardware MIMD Ray Traversal. In *Proceedings of High-Performance Graphics* (2015), pp. 21–32. 18

[LSL*13] LEE W.-J., SHIN Y., LEE J., KIM J.-W., NAH J.-H., JUNG S., LEE S., PARK H.-S., HAN T.-D.: SGRT: A Mobile GPU Architecture for Real - time Ray Tracing. In *Proceedings of High-Performance Graphics* (2013), pp. 109–119. 18, 20

[LSMY19] LIN D., SHKURKO K., MALLETT I., YUKSEL C.: Dual-Split Trees. In *Proceedings of Symposium on Interactive 3D Graphics and Games* (2019). 13

[LSS18] LIER A., STAMMINGER M., SELGRAD K.: CPU-style SIMD Ray Traversal on GPUs. In *Proceedings of High-Performance Graphics* (2018), pp. 7:1–7:4. 10, 16, 17, 23

[LV16] LIKTOR G., VAIDYANATHAN K.: Bandwidth-efficient BVH Layout for Incremental Hardware Traversal. In *Proceedings of High-Performance Graphics* (2016), pp. 51–61. 8, 18, 19

[LVY*20] LIN D., VASIOU E., YUKSEL C., KOPTA D., BRUNVAND E.: Hardware-Accelerated Dual-Split Trees. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020). 13

[LXY19] LIU Y., XU K., YAN L.-Q.: Adaptive BRDF-Oriented Multiple Importance Sampling of Many Lights. *Computer Graphics Forum 38*, 4 (2019), 123–133. 21

[LY20] LIN D., YUKSEL C.: Real-Time Stochastic Lightcuts. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 1 (2020). 21

[LYM07] LAUTERBACH C., YOON S.-E., MANOCHA D.: Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing. In *Proceedings of Symposium on Interactive Ray Tracing* (2007), p. 19–26. 13

[LYTM06] LAUTERBACH C., YOON S.-E., TUFT D., MANOCHA D.: RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of Symposium on Interactive Ray Tracing* (2006), pp. 39–46. 11

[LYTM08] LAUTERBACH C., YOON S.-E., TANG M., MANOCHA D.: ReduceM: Interactive and Memory Efficient Ray Tracing of Large Models. *Computer Graphics Forum* (2008). 13

[MB90] MACDONALD D., BOOTH K.: Heuristics for Ray Tracing Using Space Subdivision. *The Visual Computer 6*, 3 (1990), 153–65. 3, 22

[MB16] MEISTER D., BITTNER J.: Parallel BVH Construction Using k-means Clustering. *Visual Computer (Proceedings of Computer Graphics International) 32*, 6-8 (2016), 977–987. 5

[MB18a] MEISTER D., BITTNER J.: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics 24*, 3 (2018), 1345–1353. 6, 8, 9, 23

[MB18b] MEISTER D., BITTNER J.: Parallel Reinsertion for Bounding Volume Hierarchy Optimization. *Computer Graphics Forum (Proceedings of Eurographics) 37*, 2 (2018), 463–473. 7

[MBGB20] MEISTER D., BOKSANSKY J., GUTHE M., BITTNER J.: On Ray Reordering Techniques for Faster GPU Ray Tracing. In *Proceedings of Symposium on Interactive 3D Graphics and Games* (2020). 17

[MC19] MOREAU P., CLARBERG P.: *Importance Sampling of Many Lights on the GPU*. Apress, 2019, pp. 255–283. 21

[McC14] MCCOMBE J.: Introduction to PowerVR Ray Tracing. In *Proceedings of Game Developers Conference* (2014). 9, 20

[MCSM18] MAJERCIK A., CRASSIN C., SHIRLEY P., MCGUIRE M.: A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering. *Journal of Computer Graphics Techniques 7*, 3 (2018), 66–81. 15

[Mei18] MEISTER D.: *Bounding Volume Hierarchies for High-Performance Ray Tracing*. Ph.D. thesis, Department of Computer Graphics and Interaction, Faculty of Electrical Engineering, Czech Technical University in Prague, 2018. 6, 7

[Mic20] MICROSOFT: DirectX Raytracing (DXR) Functional Spec, 2020. URL: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html. 20, 21

[Mor66] MORTON G.: *A Computer Oriented Geodetic Database and a New Technique in File Sequencing*. Tech. rep., 1966. 6

[Mor11] MORA B.: Naive Ray-tracing: A Divide-and-conquer Approach. *ACM Transactions on Graphics 30*, 5 (2011), 117:1–117:12. 13

[MPC19] MOREAU P., PHARR M., CLARBERG P.: Dynamic Many-Light Sampling for Real-Time Ray Tracing. In *Proceedings of High-Performance Graphics (Short Papers)* (2019). 21

[MUWP19] MORRICAL N., USHER W., WALD I., PASCUCCI V.: Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing, 2019. 22

[MW06] MAHOVSKY J., WYVILL B.: Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum 25*, 2 (2006), 173–182. 12

[NIDN13] NABATA K., IWASAKI K., DOBASHI Y., NISHITA T.: Efficient Divide-and-conquer Ray Tracing Using Ray Sampling. In *Proceedings of High-Performance Graphics* (2013), pp. 129–135. 13

[NKK*14] NAH J.-H., KWON H.-J., KIM D.-S., JEONG C.-H., PARK J., HAN T.-D., MANOCHA D., PARK W.-C.: RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices. *ACM Transactions on Graphics 33*, 5 (2014), 162:1–162:15. 9, 20

[NKP*15] NAH J., KIM J., PARK J., LEE W., PARK J., JUNG S., PARK W., MANOCHA D., HAN T.: HART: A Hybrid Architecture for Ray Tracing Animated Scenes. *IEEE Transactions on Visualization and Computer Graphics 21*, 3 (2015), 389–401. 9, 20

[NM14] NAH J.-H., MANOCHA D.: SATO: Surface Area Traversal Order for Shadow Ray Tracing. *Computer Graphics Forum 33*, 6 (2014), 167–177. 14

[NPP*11] NAH J.-H., PARK J.-S., PARK C., KIM J.-W., JUNG Y.-H., PARK W.-C., HAN T.-D.: T&I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing. *ACM Transactions on Graphics 30*, 6 (2011), 1–10. 18, 20

[NT03] NG K., TRIFONOV B.: Automatic Bounding Volume Hierarchy Generation Using Stochastic Search Methods. 5

[NVI18] NVIDIA: *NVIDIA Turing GPU Architecture Graphics Reinvented*. Tech. rep., 2018. 20

[NVI20] NVIDIA: NVIDIA OptiX 7.2 – Programming Guide, 2020. URL: https://raytracing-docs.nvidia.com/optix7/guide/index.html. 21

[ODJ16] OGAKI S., DEROUET-JOURDAN A.: An N-ary BVH Child Node Sorting Technique for Occlusion Tests. *Journal of Computer Graphics Techniques 5*, 2 (2016), 22–37. 15, 24

[Oga16] OGAKI S.: Fragmentation-Aware BVH Contraction, 2016. URL: https://github.com/shinjiogaki/shinjiogaki.github.io. 10

[Oga20] OGAKI S.: Generalized Light Portals. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020). 21

[OHHD18] OTSU H., HANIKA J., HACHISUKA T., DACHSBACHER C.: Geometry-Aware Metropolis Light Transport. *ACM Transactions on Graphics 37*, 6 (2018). 22

[Ols18] OLSSON O.: Constructing High-Quality Bounding Volume Hierarchies for N-Body Computation Using the Acceptance Volume Heuristic. *Astronomy and Computing 22* (2018), 1–8. 22

[Pan19] PANTALEONI J.: Importance Sampling of Many Lights with Reinforcement Lightcuts Learning. *arXiv e-prints* (2019), arXiv:1911.10217. 21

[PBD*10] PARKER S., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics 29*, 4 (2010), 66:1–66:13. 15, 21

[PGDS09] POPOV S., GEORGIEV I., DIMOV R., SLUSALLEK P.: Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *Proceedings of High-Performance Graphics* (2009), pp. 15–22. 4, 5, 10, 22

[Pin10] PINTO A. S.: Adaptive Collapsing on Bounding Volume Hierarchies for Ray-Tracing. In *Proceedings of Eurographics (Short Papers)* (2010). 10, 22, 23

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation* , 3rd ed. 2016. 23

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *Proceedings of High-Performance Graphics* (2010), pp. 87–95. 6

[Ric20] RICHARD COWGILL: Introducing Ray Tracing in Unreal Engine 4, 2020. URL: https://developer.nvidia.com/blog/introducing-ray-tracing-in-unreal-engine-4. 21

[RN13] RAVICHANDRAN S., NARAYANAN P. J.: Parallel Divide and Conquer Ray Tracing. In *Proceedings of SIGGRAPH (Technical Brief)* (2013), pp. 30:1–30:4. 13

[SBK*08] SPJUT J., BOULOS S., KOPTA D., BRUNVAND E., KELLIS S.: TRaX: A Multi-Threaded Architecture for Real-Time Ray Tracing. In *Proceedings of Symposium on Application Specific Processors* (2008), pp. 108–114. 19

[SBU11] SOPIN D., BOGOLEPOV D., ULYANOV D.: Real-Time SAH BVH Construction for Ray Tracing Dynamic Scenes. In *Proceedings of Conference on Computer Graphics and Vision* (2011). 5

[SE10] SEGOVIA B., ERNST M.: Memory Efficient Ray Tracing with Hierarchical Mesh Quantization. In *Proceedings of Graphics Interface* (2010), pp. 153–160. 12, 13

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of the High-Performance Graphics* (2009), pp. 7–13. 10, 22, 23

[SGK*17] SHKURKO K., GRANT T., KOPTA D., MALLETT I., YUKSEL C., BRUNVAND E.: Dual Streaming for Hardware-accelerated Ray Tracing. In *Proceedings of High-Performance Graphics* (2017), pp. 12:1–12:11. 19

[Sil20] SILICONARTS: Ray Tracing of GPU Technology, 2020. Online Accessed: November 3rd 2020. URL: https://siliconarts.com/ray-tracing-of-gpu-technology/. 9, 20

[SKAZ19] SHYSHKOVTSOV O., KARMALSKY S., ARCHARD B., ZHDAN D.: Exploring the Ray Traced Future in Metro Exodus. In *Game Developer's Conference (GDC) 2019* (2019). 20

[SKBD12] SPJUT J., KOPTA D., BRUNVAND E., DAVIS A.: A Mobile Accelerator Architecture for Ray Tracing. In *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)* (2012). 19

[SKKB09] SPJUT J., KENSLER A., KOPTA D., BRUNVAND E.: TRaX: A Multicore Hardware Architecture for Real-time Ray Tracing. *Trans. Comp.-Aided Des. Integ. Cir. Sys. 28*, 12 (2009), 1802–1815. 19

[Smi98] SMITS B.: Efficiency Issues for Ray Tracing. *Journal of Graphics Tools 3*, 2 (1998), 1–14. 17

[SMSF20] STRÖTER D., MUELLER-ROEMER J., STORK A., FELLNER D. W.: OLBVH: Octree Linear Bounding Volume Hierarchy for Volumetric Meshes. *The Visual Computer 36*, 10 (2020), 2327–2340. 22

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: SaarCOR: A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2002), p. 27–36. 20

[SWW*04] SCHMITTLER J., WOOP S., WAGNER D., PAUL W. J., SLUSALLEK P.: Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of Graphics Hardware* (2004), pp. 95–106. 20

[TH19] TOKUYOSHI Y., HARADA T.: Hierarchical Russian Roulette for Vertex Connections. *ACM Transactions on Graphics 38*, 4 (2019), 36:1–36 : 12. 22

[Tsa09] TSAKOK J.: Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. In *Proceedings of High-Performance Graphics* (2009), pp. 151–158. 16

[Tut20] TUTTLE W.: Defining the Next Generation: An Xbox Series X|S Technology Glossary, 2020. Online Accessed: November 2nd 2020. URL: https://news.xbox.com/en-us/2020/03/16/xbox-series-x-glossary/. 20

[VAMS16] VAIDYANATHAN K., AKENINE MÖLLER T., SALVI M.: Watertight Ray Traversal with Reduced Precision. In *Proceedings of High-Performance Graphics* (2016), pp. 33–40. 18, 20

[VBH17] VINKLER M., BITTNER J., HAVRAN V.: Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction. In *Proceedings of High-Performance Graphics* (2017). 7, 23

[VBHH13] VINKLER M., BITTNER J., HAVRAN V., HAPALA M.: Massively Parallel Hierarchical Scene Processing with Applications in Rendering. *Computer Graphics Forum 32*, 8 (2013), 13–25. 13

[VHB16] VINKLER M., HAVRAN V., BITTNER J.: Performance Comparison of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing. *Computer Graphics Forum* (2016). 2

[VHBS16] VINKLER M., HAVRAN V., BITTNER J., SOCHOR J.: Parallel On-Demand Hierarchy Construction on Contemporary GPUs. *IEEE Transactions on Visualization and Computer Graphics 22*, 99 (2016), 1886–1898. 5

[VHS12] VINKLER M., HAVRAN V., SOCHOR J.: Visibility Driven BVH Build Up Algorithm for Ray Tracing. *Computers and Graphics 36*, 4 (2012), 283–296. 4

[VKJ*15] VIITANEN T., KOSKELA M., JÄÄSKELÄINEN P., KULTALA H., TAKALA J.: MergeTree: A HLBVH Constructor for Mobile Systems. In *Proceedings of SIGGRAPH Asia (Technical Brief)* (2015), pp. 12:1–12:4. 9

[VKJ*17a] VIITANEN T., KOSKELA M., JÄÄSKELÄINEN P., KULTALA H., TAKALA J.: MergeTree: A Fast Hardware HLBVH Constructor for Animated Ray Tracing. *ACM Transactions on Graphics 36*, 5 (2017). 9

[VKJ*17b] VIITANEN T., KOSKELA M., JÄÄSKELÄINEN P., IMMONEN K., TAKALA J.: Fast Hardware Construction and Refitting of Quantized Bounding Volume Hierarchies. *Computer Graphics Forum 36*, 4 (2017), 167–178. 9

[VKJ*18] VIITANEN T., KOSKELA M., JÄÄSKELÄINEN P., TERVO A., TAKALA J.: PLOCTree: A Fast, High-Quality Hardware BVH Builder. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (2018), 35:1–35:19. 9

[VKJT16] VIITANEN T., KOSKELA M., JÄÄSKELÄINEN P., TAKALA J.: Multi Bounding Volume Hierarchies for Ray Tracing Pipelines. In *Proceedings of SIGGRAPH Asia (Technical Brief)* (2016), pp. 8:1–8:4. 14, 18

[VSBY19] VASIOU E., SHKURKO K., BRUNVAND E., YUKSEL C.: Mach-RT: A Many Chip Architecture for Ray Tracing. In *Proceedings of High-Performance Graphics - Short Papers* (2019). 19

[VSBY20] VASIOU E., SHKURKO K., BRUNVAND E., YUKSEL C.: Mach-RT: A Many Chip Architecture for HighPerformance Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics* (2020), 1–1. 19

[VSM*18] VASIOU E., SHKURKO K., MALLETT I., BRUNVAND E., YUKSEL C.: A Detailed Study of Ray Tracing Performance: Render Time and Energy Cost. *The Visual Computer 34* (2018). 19

[VWB19] VAIDYANATHAN K., WOOP S., BENTHIN C.: Wide BVH Traversal with a Short Stack. In *Proceedings of High-Performance Graphics* (2019). 17

[WAB17] WOOP S., ÁFRA A., BENTHIN C.: STBVH: A Spatial-temporal BVH for Efficient Multi-segment Motion Blur. In *Proceedings of High-Performance Graphics* (2017), pp. 8:1–8:8. 11

[WAB18] WALD I., AMSTUTZ J., BENTHIN C.: Robust Iterative Find-next-Hit Ray Traversal. In *Proceedings of the Symposium on Parallel Graphics and Visualization* (2018), pp. 25–32. 15, 24

[WABG06] WALTER B., ARBREE A., BALA K., GREENBERG D. P.: Multidimensional Lightcuts. *ACM Transactions on Graphics 25*, 3 (2006), 1081–1088. 21

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 20

[Wal07] WALD I.: On Fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing* (2007), pp. 33–40. 4, 5, 10

[Wal12] WALD I.: Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 1 (2012), 47–57. 5

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs. In *Symposium on Interactive Ray Tracing* (2008), pp. 49–57. 10, 14, 22, 23

[WBKP08] WALTER B., BALA K., KULKARNI M., PINGALI K.: Fast Agglomerative Clustering for Rendering. In *Proceedings of Symposium on Interactive Ray Tracing* (2008), pp. 81–86. 5, 6, 21

[WBMS05] WILLIAMS A., BARRUS S., MORLEY R. K., SHIRLEY P.: An Efficient and Robust Ray-Box Intersection Algorithm. In *Proceedings of SIGGRAPH (Courses)* (2005), p. 9–es. 15

[WBS03] WALD I., BENTHIN C., SLUSALLEK P.: Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 77–86. 11, 23

[WBW*14] WOOP S., BENTHIN C., WALD I., JOHNSON G. S., TABELLION E.: Exploiting Local Orientation Similarity for Efficient Ray Traversal of Hair and Fur. In *Proceedings of High-Performance Graphics* (2014), pp. 41–49. 12

[WFA*05] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P.: Lightcuts: A Scalable Approach to Illumination. *ACM Transactions on Graphics 24*, 3 (2005), 1098–1107. 21

[WG16] WODNIOK D., GOESELE M.: Recursive SAH-based Bounding Volume Hierarchy Construction. In *Proceedings of Graphics Interface* (2016), pp. 101–107. 5

[WG17] WODNIOK D., GOESELE M.: Construction of Bounding Volume Hierarchies with SAH Cost Approximation on Temporary Subtrees. *Computers and Graphics* (2017). 5

[WGS04] WALDYZ I., GÜNTHERY J., SLUSALLEKY P.: Balancing Considered Harmful — Faster Photon Mapping using the Voxel Volume Heuristic. *Computer Graphics Forum 23*, 3 (2004), 595–603. 22

[WH06] WALD I., HAVRAN V.: On Building Fast KD-Trees for Ray Tracing, and on Doing that in O(N log N). In *Proceedings of Symposium on Interactive Ray Tracing* (2006), pp. 61–69. 10, 22

[Whi80] WHITTED T.: An Improved Illumination Model for Shaded Display. *Communications of the ACM 23*, 6 (1980), 343–349. 1

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes Using Coherent Grid Traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493. 2

[WIP08] WALD I., IZE T., PARKER S.: Fast, Parallel, and Asynchronous Construction of BVHs for Ray Tracing Animated Scenes. *Computers and Graphics 32*, 1 (2008), 3–13. 11

[WJLV19] WON-JONG LEE G. L., VAIDYANATHAN K.: Flexible Ray Traversal with an Extended Programming Model. In *Proceedings of SIGGRAPH Asia (Technical Brief)* (2019). 11

[WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Proceedings Eurographics Symposium on Rendering* (2006), pp. 139–149. 13

[WKB12] WALTER B., KHUNGURN P., BALA K.: Bidirectional Lightcuts. *ACM Transactions on Graphics 31*, 4 (2012). 21

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Graphics Hardware* (2006). 13, 20

[WMZ*20] WALD I., MORRICAL N., ZELLMANN S., MA L., USHER W., HUANG T., PASCUCCI V.: Using Hardware Ray Transforms to Accelerate Ray/Primitive Intersections for Long, Thin Primitive Types. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020). 12

[Wod19] WODNIOK D. M.: *Higher Performance Traversal and Construction of Tree-Based Raytracing Acceleration Structures*. PhD thesis, Technische Universität, 2019. 5

[Woo06] WOOP S.: *DRPU: A Programmable Hardware Architecture for Real-time Ray Tracing of Coherent Dynamic Scenes*. Tech. rep., Saarland University, 2006. 9, 18, 20

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444. 20

[WSWG13] WODNIOK D., SCHULZ A., WIDMER S., GOESELE M.: Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays. In *Eurographics Symposium on Parallel Graphics and Visualization* (2013). 8, 24

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G., ERNST M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics 33* (2014). 10, 14, 15, 21, 22, 23, 24

[YCM07] YOON S.-E., CURTIS S., MANOCHA D.: Ray Tracing Dynamic Scenes using Selective Restructuring. In *Proceedings of Eurographics Symposium on Rendering* (2007), pp. 73–84. 11

[YKL17] YLITIE H., KARRAS T., LAINE S.: Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In *Proceedings of High-Performance Graphics* (2017), pp. 4:1–4:13. 10, 12, 16, 17, 22, 23

[YM06] YOON S.-E., MANOCHA D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum* (2006). 8

[YS15] YTTERLID R., SHELLSHEAR E.: BVH Split Strategies for Fast Distance Queries. *Journal of Computer Graphics Techniques 4*, 1 (2015), 1–25. 22

[Yuk19] YUKSEL C.: Stochastic Lightcuts. In *Proceedings of High-Performance Graphics (Short Papers)* (2019). 21

[Zac02] ZACHMANN G.: Minimal Hierarchical Collision Detection. In *Proceedings of the Symposium on Virtual Reality Software and Technology* (2002), p. 121–128. 13

[ZHL19] ZELLMANN S., HELLMANN M., LANG U.: A Linear Time BVH Construction Algorithm for Sparse Volumes. In *Proceedings of Pacific Visualization Symposium* (2019), pp. 222–226. 22

[ZU06] ZUNIGA M. R., UHLMANN J. K.: Ray Queries with Wide Object Isolation and the DE-Tree. *Journal of Graphics Tools 11*, 3 (2006), 27–45. 13