Implementation of parallel BVH tree construction algorithms on GPU

JAN KLIMCZAK, Jagiellonian University, Poland

Bounding volume hierarchies (BVHs) are one of the most commonly used acceleration structures in computer graphics, physics simulations, and other areas. The scope of this work is to provide a description of three already existing algorithms along with their implementations and benchmark on popular datasets. The first strategy, based on Morton codes, aims to optimize construction time. The second one, based on the Surface Area Heuristic (SAH), aims to optimize query time. The hybrid solution tries to combine them, achieving both fast construction time and hierarchy optimized for ray tracing and other usages. Research on such structures is essential as the amount of data used in modern usages is growing quickly, causing parallelism to be more practical in this task.

1 INTRODUCTION

1.1 General informations

Bounding volume hierarchies are tree structures used to represent sets of geometric objects. Every node of the tree represents a bounding volume enclosing a subset of given geometric objects. Usually, but not necessarily, a single leaf corresponds to a single object. Internal nodes enclose all objects of their children nodes.

Given a set S of d-dimensional geometric objects and rooted tree T, we provide a few definitions:

- Every $v \in T$ is associated with set $S_v \subseteq S$. We say that v covers S_v.
- Every $v \in T$ is associated with geometric shape $Vol_v \subseteq \mathbb{R}^d$. We say that Vol_v is the bounding volume of v.

We say that T is a BVH tree built upon set S of d-dimensional geometric objects if the following conditions hold:

- T is a full binary, rooted tree. A full binary tree is a tree where every node has either zero or two children
- $\forall_{v \in T}\ S_v \neq \emptyset$
- If x and y are children of node v, then $S_v = S_x \cup S_y$ and $S_x \cap S_y = \emptyset$
- $S_{root} = S$
- $\forall_{v \in T} \forall_{o \in S_v} o \subseteq Vol_v$

For simplicity, we assume that all bounding volumes are AABB (axis-aligned bounding boxes). Notice that computing AABB for a single object or finding the union of two AABBs are trivial tasks.

The most common use case of BVHs is searching for intersections of a given ray with the closest object. Having this task in mind, optimal BVHs should have a few important properties:

- Overlap area of different nodes' bounding volumes should be minimal
- Node should cover objects that are relatively close to each other
- Depth of the tree should be minimal

BVHs may seem similar to other structures, such as k-d trees, but the main difference is that k-d trees are part of space division structures, which means that every node is connected with a given region instead of a set of objects. In k-d trees, a single object may be present in many leaves' sets, but areas covered by different leaves do not intersect. In BVHs, a single object is present in only one leaf's set, but areas covered by different leaves can intersect.

1.2 Usages

As mentioned, a ray-geometry intersection is the main query BVHs can solve. The recursive procedure for this task is straightforward: Starting at the tree's root, we check if its bounding volume intersects with the given ray. If not, we can discard its whole subtree. In the second case, we can recursively check for intersections in their subtrees and return the one closest to the ray origin. This simple operation can be used to implement more complicated tasks, such as ray tracing or collision detection.

Given geometric objects are often dynamic, they can change positions or shapes, and new objects may be created. To address these challenges, various methods were invented:

- BVHs with operations of inserting and deleting objects,
- grouping objects by stability, static objects can be safely stored in a single BVH, while dynamic ones should be divided into more groups to avoid a complete rebuild,
- BVHs' hierarchy staleness,

and others. In this paper, we will focus only on algorithms for constructing BVHs. The hybrid algorithm aims to achieve both quick construction time and optimized hierarchy. With these two properties, it can be safely used in real-time applications.

1.3 GPU computing model

This work is not only focused on the theoretical complexity of proposed algorithms, but it also takes into consideration consts optimizations. For it to be the most practical, we define a model that strongly resembles current GPU architectures, programming APIs, and frameworks, such as CUDA, OpenCL, GLSL, and others. To be more general, our algorithm analysis is not connected with any particular of these tools, and standard naming and definitions will be used.

With these in mind, we would like to set up our model accordingly. We define kernel as a function that runs in a parallel model. A single kernel runs on multiple threads that are grouped into thread blocks. Threads within one block can access on-chip shared memory, and they can be synchronized. However, there are limits on the number of threads in a single thread block as well as on the amount of memory a single thread block possesses. The number of threads that can run in parallel is usually much higher than the maximum number of threads in one block, and even more threads can be scheduled to run. Synchronization between different thread blocks is possible, but it is costly, the same as access to global memory; access to shared memory is usually a few hundred times faster.

Due to all this, the optimal algorithm should achieve these goals:

- Threads reading or writing to the same memory should be located in the same thread block.
- Threads should minimize accesses to the global memory and make the best use of shared memory.
- Work performed on a single thread should be small enough to avoid situations where all threads except a few are finished, and the last threads have a lot of remaining work.
- Number of threads should be large enough to utilize the full potential of parallelism and hide the latency of global memory access. Ideally, all cores should be occupied at every moment.

To analyze the performance of algorithms, standard naming will be used. The work complexity of the algorithm is the total amount of work performed by all the threads. Depth complexity is the longest chain of computations, such that for any two consecutive tasks in the chain, the first must be completed before the second can be started. Occupancy is the ratio of used threads to the maximal number of threads that can be run in parallel.

2 LINEAR BOUNDING VOLUME HIERARCHIES

Linear Bounding Volume Hierarchies (LBVHs) are a special case of BVHs. They work under the assumption that the order of the leaves is known from the start. While it may seem unlogical, various methods proved that this assumption dramatically simplifies construction algorithms while still producing fairly optimal hierarchies.

2.1 Morton Codes

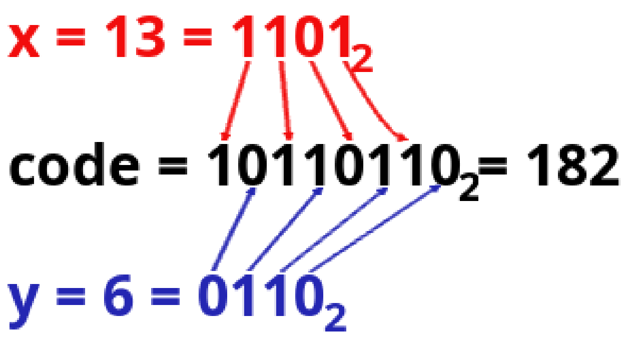

Fig. 1. Operation of interleaving bits for two-dimensional point.

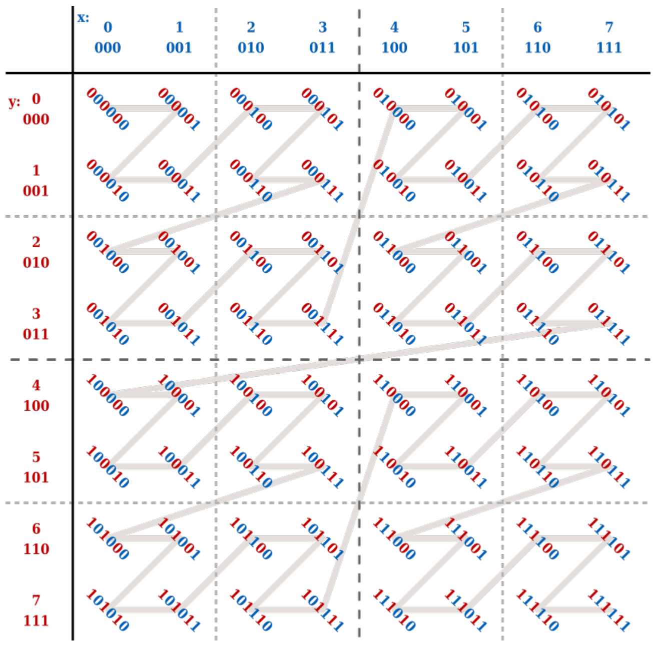

Fig. 2. The Z plane filling curve formed by interleaving the binary representations of the coordinates (x, y) and sorting the resulting interleaved binary numbers. Nomen4Omen, Public domain, via Wikimedia Commons

In a theoretical sense, a space-filling curve is a curve whose limit completely covers a predefined region, typically unit square or higher-dimension equivalents. A more intuitive definition is that a space-filling curve is a curve that produces linear ordering for cells of a regular grid. The Cantor pairing function is one example, as it maps a pair of natural numbers into a single natural number. Curves that preserve the locality of points are particularly interesting. Preserving locality means that points close to each other are mapped to numbers with small difference. Hilbert or Peano curves are good examples.

Morton code (also called the Z-order curve) is another space-filling curve. Morton code for a given two-dimensional point (with positive integer coordinates) is constructed by interleaving bits of its coordinates, as shown in Fig 1. The procedure for higher dimensional data is analogous. It can be seen that two-dimensional data is ordered along a curve resembling the 'Z' letter shape. It can be seen that Morton codes preserve locality.

Knowing the properties of space-filling curves basic idea of the algorithm emerges:

(1) Compute center for every geometric object.
(2) Compute Morton codes of produced centers.
(3) Sort geometric objects by the produced code.
(4) Generate the BVH tree under the assumption that the introduced order of objects will be the final order of the tree leaves.

To construct the first split of the tree, we can select the most significant bit that differentiates some two Morton codes; we will assign those objects with 0 bit to the left child and the rest to the right child. Then, we will recursively invoke the same procedure in both children. The produced tree can be seen as a radix tree of inputs' Morton codes.

2.2 Parallel LBVH construction

First, notice that our algorithm initially constructs Morton codes and sorts the objects by these codes. This already gives us a lower bound on the time complexity. Therefore, we aim to provide an algorithm with no more than $O(n \cdot log(n))$ work complexity and $O(log(n))$ depth.

We can transform our recursive algorithm into a parallel one. Let's notice that splits on the single level of the tree can be executed in parallel. This simple idea already has two significant drawbacks: 1) we would need a parallel work queue to store active splits, which is not a trivial task, and 2) this algorithm has critically low occupancy on the first levels of the tree. With modern GPU architectures, it would reach full occupancy after several levels of the tree.

Various methods of solving this problem were invented, and we present two of them.

2.3 Calculating bounding boxes

Before we present any of the algorithms, we will solve a simpler problem that will be used in both. Suppose that we have already produced the tree structure. What is left to do is to calculate a valid bounding volume for every node. Finding the smallest bounding box for a single geometric object is trivial. Therefore, it is easy to find bounding boxes for leaves, as they consist of single objects. Finding a bounding box for the internal node given boxes of its children is also easy.

We will assign an atomic counter with standard increment operation for every node. Then we will invoke an identical kernel on all nodes in parallel. The kernel calculates the leaf's bounding box, traverses to the parent node, and increments its counter. If a thread was the first one to increment the counter, it is terminated. If a thread was the second one to do it, we are sure that the bounding boxes for children are already calculated. Therefore it can calculate the bounding box for the given node and move up the tree again.

It is worth noting that this algorithm will have low occupancy at the upper levels of the tree, but it is acceptable since it is linear in terms of work and the depth is logarithmic.

2.4 C. Lauterbach et al.

The main idea behind the first presented method [Lauterbach C. 2009] is to build a tree that is not optimal in terms of depth and optimize it afterward. Let's notice that the trie of given Morton codes is almost a valid BVH tree, the only problem is that it is not necessarily a full binary tree. However, it has an important property - the Morton code of the leaf node is equivalent to labels on the path from the root to the considered node. Therefore, given two consecutive leaves with indices i and $i + 1$, their lowest common ancestor can be computed by taking the longest common prefix of their Morton codes' bits.

Knowing this, we can compute common prefixes for every consecutive pair of indices. For the given pair, let h denote the length of the prefix found and let k denote the length of the binary representation of Morton codes. We will produce a list of pairs $((h + 1, i), (h + 2, i), ...(k, i))$ as the output for the given pair of consecutive indices. This list can be seen as a list of levels at which leaves i and $i + 1$ are in separate subtrees. We can simultaneously produce similar lists for every consecutive pair of indices and then concatenate them. After that, resort the list by the first value in the pair - level in the tree. Let's notice that this list explicitly tells us, for each level, how many nodes there are and what their ranges are. It is also possible to calculate indices of children nodes for any particular node - knowing that on level i, there is a node that covers a range of leaves $[l, r]$, its left child has to be a node on level $i + 1$ that covers a range of leaves starting at l, similarly for the right child. Therefore, such nodes can be found using binary search. Knowing this, we can generate all nodes in parallel.

One drawback to this method is that it produces long chains of nodes since all codes in some node's range can have the same most significant bit. We can resolve this by introducing an additional step after constructing the initial hierarchy. Let's invoke an identical kernel on all leaves in parallel. A single thread traverses the tree from the leaf to the root. In any particular node, if it has only one child, the thread replaces all of the node's attributes with respective values from the child node. The kernel does nothing in the second case, i.e., when the node has two children. Afterward, the kernel traverses to the parent of the current node. We can reuse the traversal method from the previously described kernel that calculated bounding boxes. However, it is also worth noting that the current kernel performs no operations in nodes with two children. Therefore it does not rely on previously calculated values, so it is enough to terminate the thread from the right child.

This already gives us optimal structure, which can be seen as the radix tree of the input.

2.5 Karras

The previous method overcomes the difficulty by constructing not optimized tree first, which turned out to be a simpler task in a parallel setting. The second method introduces a different scheme of labeling internal nodes of the tree, allowing to process all of them simultaneously [Tero Karras 2012].

The first observation is that a binary tree with n leaves has exactly $n - 1$ internal nodes. Let's index leaves with indices from 0 to $n - 1$ and internal nodes with indices from 0 to $n - 2$. Invoke procedure for all internal nodes in parallel:

(1) Let range covered by node v be $[l, r]$. Find the most significant bit that differs in some two leaves in the given range. A crucial observation is that it is enough to find the first bit that differs in the first and the last leaves.
(2) Using binary search, find last position i, such that leaves with indices i and r differ at previously found bit. This will be the split position, it follows that children of v cover ranges $[l, i]$ and $[i + 1, r]$ respectively.
(3) If i is equal to l (so that range $[l, i]$ has only one element), assign l-th leaf node as left child of v. Analogously, if i is equal to r, assign r-th leaf node as the right child.
(4) If i is not equal to l, assign i-th internal node as left child of v. Analogously if $i + 1$ is not equal to r, assign $i + 1$-th internal node as the right child. It is worth noting that children of every node will have consecutive indices.

The sample output can be seen in Figure 3.

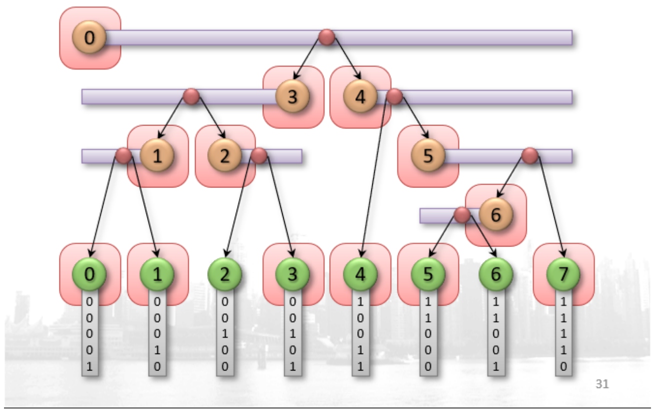

Fig. 3. Tree produced with the second algorithm. Siblings in the tree have consecutive indices, and the index of every node is equal to one of its range's ends

.

The given procedure is still incomplete, as in the first step, we assumed that we know the range covered by the node, given only its index. Suppose for a moment that we can calculate it, and let's prove that the produced labeling is proper, i.e., no two internal nodes share the same index. Notice that indices of children of a given node lie in the range it covers, and the ranges of its children do not overlap. Thus, it is enough to show that a node with index i cannot have another node with the same index as its descendant. Suppose a node with index i covers range $[i, r]$ (so it was the right child in the split). For another node with index i to be created, a split between indices i and $i+1$ or $i-1$ and i must occur. The second case is impossible because $i - 1$ is out of the node's range. So in the first case, if there occurs a split between indices i and $i + 1$, ranges of the produced children nodes are $[i, i]$ and $[i + 1, r]$ respectively, but since $[i, i]$ is a range with only one element, the left child of a given node will be a leaf node and not an internal node. This concludes the proof.

To finish the algorithm, two simple observations are needed:

- Internal node with index i covers range of form $[l, i]$ or $[i, r]$.
- Any two leaves in some node's range have more leading bits in common with each other than with any leaf outside the range. This statement is trivial in tries or radix trees, and our structure is a radix tree.

These two facts are enough to complete the algorithm. First, check if node i covers the range in form $[l, i]$ or $[i, r]$. To do this, check how many common leading bits leaf i has with leaves $i-1$ and $i+1$. Let these values be x and y, respectively. Assume that $x < y$ for the rest of the algorithm. Therefore, the range of node i cannot have the form $[l, i]$ because $i-1$-st leaf would be inside it and $i+1$-th leaf would be outside, which contradicts the second observation. It follows that the range is of form $[i, r]$

Range of i-th node's father was of the form $[l, r]$ for some l, r satisfying $l < i \leq r$, and split divided this range into $[l, i-1]$ and $[i, r]$. From the second observation, the i-th leaf has to have more bits in common with r-th leaf than with $(i-1)$-th leaf. It must also have more bits in common with $(i-1)$-th leaf than with $(r+1)$-th leaf. So to find the r, it is enough to find the first index j, such that the number of common bits of i-th and j-th leaves is less than x. This can be achieved with binary search.

The most costly operations in the algorithm are two binary searches performed for every node, so work complexity is again $O(n \cdot log(n))$.

3 SURFACE AREA HEURISTIC

3.1 Idea

LBVHs worked under the assumption that the order of the leaves is known from the start of the algorithm. This led to efficient methods of constructing hierarchies, but these hierarchies are not optimized for ray tracing, as objects that got similar Morton codes may be far from each other. Therefore, SAH (Surface Area Heuristic) was introduced [Goldsmith J. 1987]. Note that the assumption about leaves' order does not apply now.

The main idea is as follows: assume, for each node, that its children are leaves, probably containing more than one geometric object. Calculate the expected number of ray-geometry intersections that need to be performed for random ray intersecting this node's bounding box. More formally, the cost of a node v is defined as

$$Cost(V) = K_T + K_I \cdot \left(\frac{SA(V_L)}{SA(V)} \cdot N_L + \frac{SA(V_R)}{SA(V)} \cdot N_R \right),$$

where K_T and K_I are predefined const values, estimating the relative cost of node traversal and geometry intersection test; V_L and V_R are left and right children, respectively; $SA(V)$ is the surface area of V's bounding volume; N_L and N_R are numbers of geometric objects covered by L and R respectively.

The Surface Area Heuristic aims to greedily minimize these costs recursively, i.e., choose an optimal split method for the root and then consider children nodes recursively. Most terms in the provided equation do not depend on the choice of leaves distribution between children nodes. Therefore, it can be simplified as

$$Cost(v) = (SA(V_L) \cdot N_L + SA(V_R) \cdot N_R)$$

Let's discuss a sequential algorithm first. Nodes will be processed recursively, starting from the root covering all geometric objects. We will consider some ways to split these objects into two sets, choose one minimizing the cost function, and invoke the same function on produced children. However, it is not clear how to choose the optimal split. One of the ways is to consider all of the axis-aligned separating lines/planes. We can consider every dimension separately (considering all lines/planes aligned to one particular axis simultaneously). To do this, we can sort the objects along the chosen dimension, calculate bounding boxes for each suffix and prefix of them, and finally consider every split position. This method produces highly optimized hierarchies, but it is expensive in terms of computation as it requires sorting and additional linear work to consider all splits. It is worth noting that it is possible to perform only one sorting for each axis at the beginning and reuse it later.

Another idea is to randomly choose a fixed number of candidate splits instead of considering all of them. Various studies have shown that this method produces hierarchies almost as good as full SAH computation. [Wald 2007]

3.2 Algorithm

SAH construction cannot be parallelized in the same way as previous algorithms because both of them relied strongly on the fact that the order of leaves is known. However, we could not parallelize a single split in sequential LBVH construction because there was not enough work for more threads (as a single split consisted only of two binary searches). Note that this time, the cost function must be evaluated for every split, which is a more costly operation and gives some hope for parallelizing a single split.

We will describe a method for parallelizing a randomized method (with a fixed amount of split planes), but full SAH evaluation can also be parallelized. We will consider k splits for every axis, so we will evaluate $3k$ splits in a three-dimensional case. If there are less than k objects in a given node set, we will evaluate full SAH instead. We will run $3k$ threads simultaneously for each split, and each thread will evaluate one split plane. The algorithm loads a fixed amount of objects into shared memory, and each thread tests, for every object, on which side of the split plane it lies, updating one of the bounding boxes accordingly. After considering all objects, the best split is chosen by parallel reduction. After the best split is chosen, objects have to be reordered. For each object in parallel, we will test again on which side of the optimal split plane it lies and assign 0 or 1 to it, depending on the result. Then we will perform a parallel scan of the produced array. It is then trivial to compute a new position for each element. Note that moving a large amount of data may not be optimal, and one of the possible solutions is only to move indices.

The provided algorithm utilizes parallelism, but for this algorithm to be faster than full SAH evaluation, k should be relatively small, so invoking only $3k$ threads will lead to low occupancy and will not hide memory access latency. Therefore we will also process different splits in parallel. As mentioned in the previous section, we need to implement a work queue to store all currently active splits.

In a general setting, a work queue is not feasible, but one important property of BVH construction is that every node has at most two children. So the idea is simple: we will store a queue for each level of the tree, starting with a queue containing only the root. Suppose a given level has a queue with size x and allocate an array of size $2x$ for the next level. If a node on position i in the queue produces new splits to consider, put them in positions $2i$ and $2i + 1$ in the next queue. Some nodes may produce only 1 or even 0 additional work, which will cause the next queue to have empty spaces. To eliminate them, assign 1 to every non-empty position and run a parallel scan algorithm. Calculated prefix sums can be used to reorder elements to eliminate all empty positions.

3.3 Small split optimization

The algorithm just given has two bottlenecks: 1) the top levels of the tree cannot fully utilize resources, as there are not enough threads scheduled, and 2) the bottom levels of the tree have too many context switches because work on the single split is relatively small, there are a lot of active splits, and there is a lot of time wasted on queue compaction. To address the second problem, additional optimization was introduced. [Lauterbach C. 2009] An additional queue will be introduced to store all active splits with sizes less than some fixed threshold. During the queue compaction, all small splits are filtered into this new queue, and after the main queue is empty, this new queue with small splits will be processed. For every small split, we will assign one thread group to process it entirely without creating new work items. Information about small split can fit into shared memory, significantly reducing time spent on memory accesses.

4 HYBRID ALGORITHM

Both presented algorithms have some drawbacks: 1) LBVH construction is fast and fully utilizes parallelism, but it produces hierarchies that are not optimal for queries, and 2) SAH construction lacks parallelism on the top levels of the tree but produces more accurate hierarchies. To deal with these drawbacks, we present a hybrid algorithm [Lauterbach C. 2009]: first, build the fixed number of the top levels using the LBVH algorithm and then consider all the produced leaves as active splits and build SAH optimized hierarchy on them. This algorithm fixes problems of both approaches, as it eliminates the top levels of the tree from SAH construction and aims to produce an optimized hierarchy with only a few unoptimized levels at the top.

It is worth noting that LBVH requires the input to be sorted by the object's Morton codes, but since we want to use the LBVH algorithm to construct only the first few levels, we can compute only the first few bits of Morton codes. Due to this, keys passed to the sorting algorithm have short binary representations; therefore, radix sort gets an advantage over other sorting algorithms, as it would require only a few passes to sort the sequence.

5 RESULTS

To properly measure algorithms' performance, we conducted tests on four popular models of varying complexities. Everything was performed on a single machine, with AMD Ryzen 7 5800H, 3.20 GHz, and NVIDIA GeForce RTX 3060 with 6 GB of memory. Algorithms were implemented in OpenGL Shading Language (GLSL).

We use two simplest statistics to compare results: construction time and time of rendering one frame of a static scene. Therefore, we implemented a simple version of the packet traversal ray tracing algorithm [Gunther J. 2007]. We rendered every scene in 1024×1024 resolution, casting only one primary ray for each pixel. Since few algorithms' parameters were modifiable, we have tried a few configurations, but we present these with the best results.

Since GLSL operates only on 4 byte numeric values, Morton codes consist of only 30 bits (10 bits for each axis). This can lead to performance loss in highly detailed scenes.

To evaluate SAH, we randomly choose 128 possible planes for each split, and each plane is evaluated by a separate thread. The threshold for a split to be filtered into a small queue was set to 32. As expected, the introduction of a small queue led to a performance boost of around 15% due to higher occupancy and fewer context switches. However, it turned out that ray

Model	Triangles	LBVH	SAH	Hybrid
Happy Buddha	51415	25ms 11ms/67%	77ms 8ms/100%	47ms 8ms/98%
Stanford bunny	69451	20ms 11ms/73%	85ms 8ms/100%	44ms 8ms/99%
Sponza	135394	98ms 18ms/76%	350ms 13ms/100%	130ms 13ms/98%
Hairball	2880000	617ms 67ms/73%	2742ms 51ms/100%	1143ms 52ms/98%

Table 1. First row for each model contains construction times (in ms) for all steps of BVH construction, time of copying data from CPU to GPU is omitted. As the rendering time is usually low, the second row additionally contains the relative framerate of rendering the model with 1024×1024 resolution. The fastest algorithm for each case serves as a baseline.

tracing had a similar problem on lower levels of the tree - traversing the lower levels with only a few objects in every node caused minor performance problems. Therefore, leaving nodes with less than 32 objects unsplit was the most optimal solution, and results were measured for this version.

To evaluate the hybrid algorithm, we construct the first 6 levels of the tree using the LBVH algorithm, and then we switch to the SAH algorithm with already discussed parameters.

While not the most efficient, our algorithms strongly resemble the relative performance of other known benchmarks. SAH construction may be 4 to 5 times slower than LBVH on most of the models. However, it yields more than 25% of speedup in rendering. Therefore, even for large models, the speedup in the rendering will compensate the difference in construction time in a matter of seconds. On the other hand, in fully dynamic scenes, where BVH has to be reconstructed from scratch every time, LBVHs can be more than 3 times faster to render one frame. However, the hybrid algorithm offers almost the same performance in rendering as BVH, and it is much closer to LBVH construction time. Therefore, it may be optimal for most cases besides those fully dynamic.

REFERENCES

Salmon J. Goldsmith J. 1987. *Automatic Creation of Object Hierarchies for Ray Tracing, IEEE Computer Graphics and Applications, Volume 7, Issue 5.* https://dl.acm.org/doi/10.1109/MCG.1987.276983

Seidel H. Slusallek P. Gunther J., Popov S. 2007. *Realtime Ray Tracing on GPU with BVH-based Packet Traversal.* https://www.johannes-guenther.net/BVHonGPU/BVHonGPU.pdf

Sengupta S. Luebke D. Manocha D. Lauterbach C., Garland M. 2009. *Fast BVH Construction on GPUs.* https://luebke.us/publications/eg09.pdf

NVIDIA Research Tero Karras. 2012. *Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees.* https://research.nvidia.com/sites/default/files/pubs/2012-06_Maximizing-Parallelism-in/karras2012hpg_paper.pdf

Ingo Wald. 2007. *On fast Construction of SAH-based Bounding Volume Hierarchies.* https://www.sci.utah.edu/~wald/Publications/2007/ParallelBVHBuild/fastbuild.pdf