CGI2016 manuscript No.
(will be inserted by the editor)

Parallel BVH Construction using k-means Clustering

Daniel Meister · Jiří Bittner

Abstract We propose a novel method for fast parallel construction of bounding volume hierarchies (BVH) on the GPU. Our method is based on a combination of divisible and agglomerative clustering. We use the k-means algorithm to subdivide scene primitives into clusters. From these clusters we construct treelets using the agglomerative clustering algorithm. Applying this procedure recursively we construct the entire bounding volume hierarchy. We implemented the method using parallel programming concepts on the GPU. The results show the versatility of the method: it can be used to construct medium quality hierarchies very quickly, but also it can be used to construct high quality hierarchies given a slightly longer computational times. We evaluate the method in the context of GPU ray tracing and show that it provides results comparable with other state-of-the-art GPU techniques for BVH construction. We also believe that our approach based on the k-means algorithm gives a new insight on how bounding volume hierarchies can be constructed.

Keywords Ray Tracing · Object Hierarchies · Three-Dimensional Graphics and Realism

1 Introduction

Spatial data structures such as octrees, kD-trees, or bounding volume hierarchies play an important role in computer graphics as they help to handle the ever increasing scene complexity. The bounding volume hierarchies (BVH), which we address in the paper, have been shown to accelerate a number of intersection queries, particularly in the context of collision detection and ray tracing. The BVH is a hierarchical object partitioning and thus it has a predictable memory footprint (every primitive is referenced exactly once in the data structure) and can handle dynamic scenes by refitting, i.e. a simple adaptation of the bounding volumes keeping the same hierarchy topology.

Daniel Meister
Czech Technical University in Prague
Faculty of Electrical Engineering

Jiří Bittner
Czech Technical University in Prague
Faculty of Electrical Engineering

The BVH can be constructed by partitioning the set of scene objects recursively: when constructing a binary BVH we split the current set of objects into two groups according to certain rules and continue the process until we reach termination criteria. The rule used to partition the objects can have a significant impact on the efficiency of the constructed BVH for a particular application. In the context of ray tracing the surface area heuristic [14] is commonly used to optimize the object partitioning. Another approach is to construct the BVH from bottom to top by agglomerative clustering [35]. While this approach can lead to higher quality trees (measured using SAH cost [14]), it also requires much higher computational effort. Recently, Gu et al. [15] proposed a method that combines top-down construction with local agglomerative clustering. A viable alternative for the GPU construction of high quality BVHs are the techniques that first construct the BVH using a fast Morton code based algorithm and then perform treelet restructuring to optimize its topology [22,9]. These methods allow trading quality for performance and can be tuned for a desired BVH quality.

We propose a new technique for massively parallel GPU based BVH construction that gives novel ideas into the field of BVH construction. In our approach we use the well-known k-means clustering [28] as a basis for the BVH construction. By using the k-means clustering the top-down phase of our algorithm can already compute high quality clusters which allows us to use a simple bottom-up merging procedure and in turn to implement the whole method on the GPU.

Our paper aims at the following contributions: (1) according to our knowledge we are the first to apply the k-means clustering in the context of BVH construction for ray tracing, (2) our method provides a flexible means to trade BVH quality for construction speed, (3) we show that the results of the method are comparable with the latest GPU BVH builders that use already established methods such as Morton code based primitive sorting.

2 Related Work

BVH Bounding volume hierarchy is one of the most common acceleration data structures in the context of ray tracing. Already in the early 80s Rubin and Whitted [31] used a manually created BVH, while Weghorst et al. [36] proposed to build the BVH using the modeling hierarchy. The very first BVH construction algorithm using spatial median splits was introduced by Kay and Kajiya [23]. Goldsmith and Salmon [14] proposed a cost function known as the *surface are heuristic* (SAH). This function can be used to estimate the efficiency of a BVH during its construction and thus most state of the art BVH builders are based on SAH. The BVH construction methods require sorting and thus generally exhibit $\mathcal{O}(n \log n)$ complexity (n is the number of scene triangles). Several techniques have been proposed to reduce the constants behind the asymptotic complexity. For example Havran et al. [16], Wald et al. [33], and Ize et al. [19] used approximate SAH cost function evaluation based on binning. Hunt et al. [18] suggested to use the structure of the scene graph to speed up the BVH construction process. Dammertz et al. [7] proposed to use BVHs with a higher branching factor to better exploit SIMD units in modern CPUs.

High quality BVH Recently more interest has been devoted to methods, which are not limited to the top-down BVH construction. Walter et al. [35] proposed to use bottom-up agglomerative clustering for constructing a high quality BVH. Gu et al. [15] proposed a parallel approximative agglomerative clustering for accelerating the bottom-up BVH construction. Kensler [24], Bittner et al. [5], and Karras and Aila [22] proposed to optimize the BVH by performing topological modifications of the existing hierarchy. Recently Ganestam et al. [12] introduced the Bonsai method performing a two level SAH based BVH build on a multi-core CPU. These approaches allow to decrease the expected cost of a BVH beyond the cost achieved by the traditional top-down approach. Several extensions of the basic SAH have also been proposed to increase the BVH performance for specific applications. Hunt [17] proposed corrections of SAH with respect to mailboxing. Fabianowski et al. [10] proposed SAH modification for handling scene interior ray origins. Bittner and Havran [6] proposed to modify SAH by including the actual ray distribution, Feltman et al. [11] extended this idea to shadow rays. Corrections of the SAH based BVH quality metrics have been proposed by Aila et al. [1].

Parallel BVH construction Recently both multi-core CPUs and many-core GPU construction methods of BVHs have been investigated. Lauterbach et al. [26] proposed a GPU method known as LBVH, which is based on the Morton curve and spatial median splits. Wald [34] studied the possibility of fast rebuilds from scratch on an Intel architecture with many cores. Pantaleoni and Luebke [30], Garanzha et al. [13], Karras [21], Vinkler et al. [32], and Domingues and Pedrini [9] proposed methods for parallel BVH construction that achieve impressive performance on the recent GPUs. The method of Karras [21], which was further improved by Apetrei [3] is considered a fastest available BVH builder on the GPU, but it generally builds trees of slightly lower quality. A good balance between the build time and tree quality can be achieved by the combination of a fast BVH build and subsequent treelet optimization on the GPU as proposed by Karras and Aila [22] and further optimized by Domingues and Pedrini [9]. We use LBVH [21], HLBVH [13], ATRBVH [9], and the AAC [15] methods as references for our comparisons.

The paper is further organized as follows: Section 3 describes the overview of our method, Section 4 describes the main components of the proposed method, Section 5 gives details about the GPU implementation, Section 6 provides the results and evaluation, and finally Section 7 concludes the paper.

3 Method Overview

We first introduce the basics of hierarchical clustering and then we give an outline of the proposed algorithm by describing its elementary steps.

3.1 Hierarchical Clustering

Hierarchical clustering is a well-known approach in pattern recognition, image analysis, and bioinformatics. Given an n-element set $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ and a distance function d such that $d(\mathbf{x}_i, \mathbf{x}_j) > 0$ for $i \neq j$

and $d(\mathbf{x}_i, \mathbf{x}_j) = 0$ for $i = j$, we construct a tree $\mathcal{T} = \{\mathcal{L}_1, \ldots, \mathcal{L}_m\}$ such that

- $\mathcal{L}_i$ is a partition of $\mathcal{X}$ for $i \in \{1, \ldots, m\}$,
- $\mathcal{L}_i$ is a proper refinement of $\mathcal{L}_{i+1}$ for $i \in \{1, \ldots, m-1\}$,
- $\mathcal{L}_1 = \{\{\mathbf{x}_1\}, \ldots, \{\mathbf{x}_n\}\}$,
- $\mathcal{L}_m = \{\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}\}$.

The goal is to minimize the objective function

$$\sum_{\mathcal{L}_k \in \mathcal{T}} \sum_{\mathcal{C}_i \in \mathcal{L}_k} \sum_{\mathbf{x}_j \in \mathcal{C}_i} d(\overline{\mathcal{C}_i}, \mathbf{x}_j),$$

where $\overline{\mathcal{C}_i}$ denotes the mean of cluster $\mathcal{C}_i$. Unfortunately the hierarchical clustering problem is NP-hard [25]. However there are greedy heuristics which work quite well in practice [20]. The main hierarchical clustering strategies include top-down hierarchy construction (divisible clustering) or bottom-up construction (agglomerative clustering).

One of the most popular divisible clustering algorithm is the k-means algorithm [28, 27]. Initially k cluster representatives are chosen. The common practice is to draw the representatives randomly from $\mathcal{X}$. However there are more sophisticated approaches how to choose the representatives, e.g. `k-means++` [4]. The k-means algorithm works iteratively by assigning elements $\mathbf{x} \in \mathcal{X}$ to the cluster representatives and thus forming the clusters. First, each element $\mathbf{x} \in \mathcal{X}$ is assigned to the nearest representative. Second, the cluster representatives are replaced by the mean of all elements forming the cluster. This procedure is repeated until the maximum number of iterations is reached. The cluster hierarchy is constructed by applying the k-means algorithm recursively.

Another popular hierarchical clustering method is agglomerative clustering [35]. The agglomerative clustering starts with level $\mathcal{L}_1$ and iteratively constructs higher levels. In each step algorithm finds two nearest clusters among the sibling nodes according to the function d. Then both clusters are merged together. This procedure is repeated until $\mathcal{L}_m$ is constructed.

3.2 Algorithm Outline

Our algorithm constructs the BVH by a combination of divisible and agglomerative hierarchical clustering. First we associate all scene primitives with the root node of the BVH. This root node is then subdivided into k clusters using the k-means algorithm. We use a data parallel approach so even at the top of the hierarchy the k-means clustering can be efficiently executed on the GPU. The k-means algorithm is then applied on all nodes resulting from the previous k-means execution that do not fulfill a termination criterion (number of triangles per node). Thus we build one level of a k-ary BVH at each step of the algorithm. In most cases ray tracers are optimized for BVHs with low branching factors such 2 or 4. Thus we postprocess the resulting BVH by performing agglomerative clustering within each node of the k-ary BVH. This step expands each interior node of the k-ary BVH to a treelet in a resulting binary BVH. The agglomerative clustering step is limited to the k children of each input BVH node and thus can be applied in parallel on all BVH nodes. The main steps of the algorithm are illustrated in Figure 1.

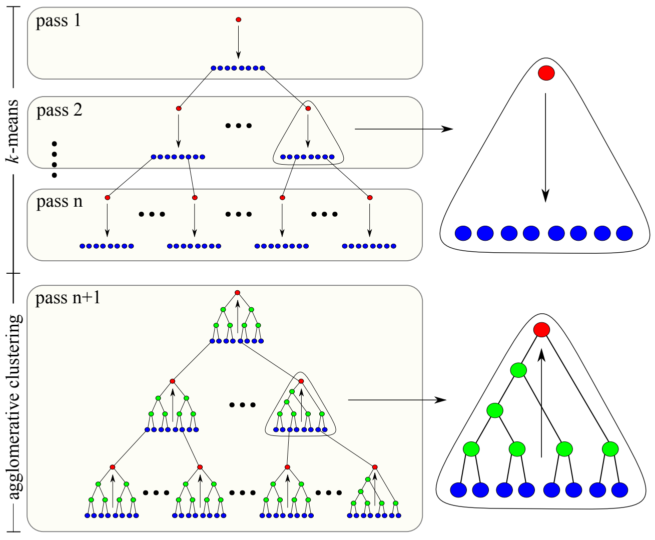


Fig. 1 Illustration of the proposed algorithm. First we use the k-means algorithm to build a k-ary BVH by sorting node primitives to clusters (blue nodes). Then we use the agglomerative clustering algorithm to build the intermediate levels of the output binary BVH (green nodes). Although depicted as a complete n-ary tree, the BVH need not be balanced in general.

4 BVH Construction using k-means with Agglomerative Clustering

In this section we first describe how to use the k-means algorithm to build a k-ary BVH. Then we describe how to apply agglomerative clustering to convert the k-ary BVH to a binary BVH commonly used for ray tracing.

4.1 k-means for BVH

The k-means algorithm needs a definition of the distance function d used for distributing the primitives to the clusters. We assume that the bounding volumes associated with BVH nodes correspond to axis aligned bounding boxes. A bounding box $\mathbf{b}$ is defined by two extreme points $\mathbf{b}^{min}$ and $\mathbf{b}^{max}$ representing the minimal and maximal coordinates of all points enclosed by $\mathbf{b}$ in all three axes. Given two bounding boxes $\mathbf{b}_1$ and $\mathbf{b}_2$, we define the distance function d as a sum of squared Euclidean distances between the extreme points of $\mathbf{b}_1$ and $\mathbf{b}_2$:

$$d(\mathbf{b}_1, \mathbf{b}_2) = ||\mathbf{b}_1^{min} - \mathbf{b}_2^{min}||^2 + ||\mathbf{b}_1^{max} - \mathbf{b}_2^{max}||^2. \quad (1)$$

The distance function thus corresponds to a squared Euclidean distance in $\mathbb{R}^6$ when considering the pair of extreme points of a given bounding box as a point in $\mathbb{R}^6$. We tried other distance functions including the well-known Manhattan and Chebyshev metrics. We also experimented with distance functions taking into account various spatial relations of bounding boxes, e.g. the surface area of union of bounding boxes [35]. However, the described distance function provided the most stable results with respect to the final BVH quality. Note, that this corresponds to what is known about k-means: they perform well for hierarchical clustering with Euclidean metrics, but need not converge with other distance metrics [8].

In the k-means algorithm we first have to initialize the cluster representatives. We use a simple heuristic to draw the initial representatives from bounding boxes of scene primitives. The first representative is drawn randomly. The i-th representative is determined by randomly drawing p candidates and choosing the one maximizing the distance to the nearest already determined representative. We have also tested the `k-means++` [4], which however shown to be too slow for our purposes.

At the core of the k-means algorithm we first assign each scene primitive to the nearest representative according to the function d. Then we update the representatives. The new representative $\mathbf{r}_i$ associated with the cluster $\mathcal{C}_i$ is the mean of bounding boxes in cluster $\mathcal{C}_i$ computed as

$$\mathbf{r}_i^{t+1} = \overline{\mathcal{C}_i^t} = \frac{1}{|\mathcal{C}_i^t|}\Big(\sum_{\mathbf{b}_j \in \mathcal{C}_i^t} \mathbf{b}_j^{min}, \sum_{\mathbf{b}_j \in \mathcal{C}_i^t} \mathbf{b}_j^{max}\Big). \quad (2)$$

The assignment to the nearest cluster and the cluster update are performed iteratively, where the number of iterations is a parameter of the method. Therefore we also refer to this part of the algorithm as the k-means loop. The whole k-ary tree is constructed by applying this procedure recursively, while each level of the tree may be processed in parallel. Note that the number of iterations in the k-means loop may be set to zero, in which case we just assign scene primitives to the initial representatives and do not update this assignment any further. An illustration of the results of the k-means clustering and the corresponding cluster representatives is shown in Figure 2.

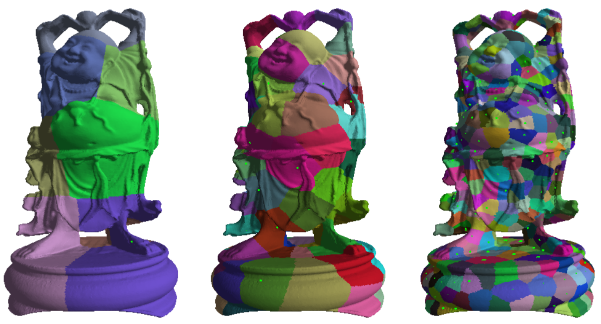


Fig. 2 Example of the results of k-means clustering ($k = 8$) for the first three passes of the algorithm using five k-means iterations. Triangles belonging to different clusters are shown in different colors. Cluster representatives are shown as axis aligned boxes in green.

4.2 Agglomerative Clustering

We use agglomerative clustering to build the intermediate levels of the tree. As we use relatively small values of k (8, 16, 32, or 64) a naïve agglomerative clustering considering all pairs of clusters provides a sufficient performance. The implementation of the naïve agglomerative clustering is simple and requires no additional data to be kept or preprocessed. In this phase of the algorithm we use the surface area of the merged cluster as a distance function between two clusters as proposed by Walter et al. [35]. The main advantage of this approach compared to previous techniques is that as we already have a k-ary BVH available we can process all treelets in parallel.

5 GPU Implementation

We implemented our BVH construction algorithm in CUDA [29]. We use a queue system proposed by Garanzha et al. [13]. The input queue is used for handling unprocessed tasks and the output queue is used for generating new tasks. At the beginning the input queue contains only one task corresponding to the root of the hierarchy and all triangles are assigned to this task. Each task may produce up to k new tasks using the k-means algorithm. Threads process data corresponding either to tasks or to triangles in parallel. Each task is associated with a continuous segment in the triangle indices array. We also use an auxiliary array storing an index of the corresponding task for each triangle. Thus we can map triangle to task and vice versa. The overview of the complete algorithm is shown in Figure 3. In the remainder of this section we provide details about the individual steps of the implementation of the proposed algorithm.

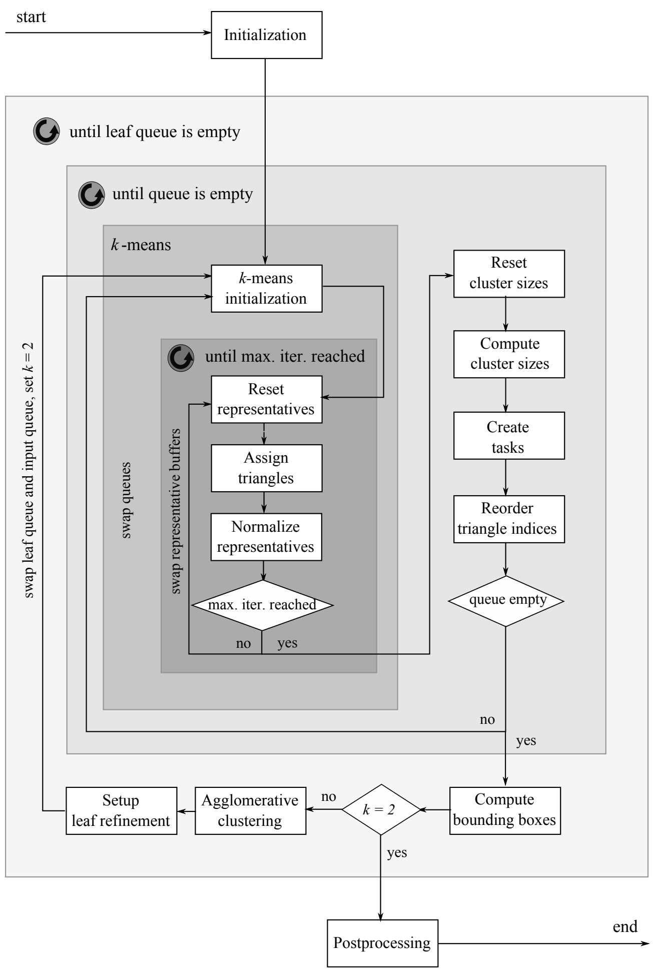


Fig. 3 Overview of the algorithm. White rectangles represent kernel launches.

Initialization The algorithm starts by computing bounding boxes and indices of triangles. Then the algorithm enters the main loop depicted in Figure 3.

k-means initialization We use two (input and output) arrays to store the cluster representatives. The very first kernel implements the previously described heuristic initializing the representatives. We use a simple linear congruential generator to draw representative candidates.

k-means loop The k-means loop consists of three kernels. The first kernel resets the cluster representatives by setting the output array to zeros. The second kernel assigns triangles to the nearest representative. Each triangle is processed by a single thread in parallel. This thread finds the nearest input representative, atomically increments the corresponding triangle counter and atomically adds the extremes of the triangle bounding box to the corresponding output representative. For higher k it may happen that some cluster representatives are equal. If this happens the triangles might be assigned to just one of these representatives. As a consequence empty nodes may occur that correspond to the other representative to which no triangle has been assigned. Thus during the triangle assignment we use a simple rule based on triangle indices to ensure that at least two clusters are not empty even when all representatives are the same. At the end of the k-means loop iteration the third kernel normalizes the representatives by dividing the output representatives by the corresponding triangle counters. At the end of the iteration the input and output representatives are swapped.

Cluster size computation Before creating new tasks corresponding to the new clusters we have to determine cluster sizes. We run a very similar kernel to the one that we used for assigning triangles in the k-means loop. For each triangle we determine the nearest representative and atomically increment the corresponding triangle counter. To avoid redundant computations we store an index of the nearest representative in an auxiliary array.

Task creation When we know the number of triangles in each cluster, we can create new tasks. Let m denote the maximum leaf size. If the cluster contains less than m triangles, it is marked as leaf, otherwise it is handled as a new interior node and a corresponding new task is created. If the interior node contains less than mk triangles and $k > 2$ the task is put to the auxiliary leaf queue that handles the treelets at the bottom of the tree. Otherwise the new task is put to the output queue. We use the approach proposed Garanzha et al. [13] to determine the positions of new task in the output queue. Threads compute the number of output tasks and then a warp-wide prefix scan is performed. The first thread within the warp atomically adds the number of output tasks to the global counter. Atomic addition returns the original value of the counter which is the offset for threads within the warp. We also perform (sequential) exclusive prefix scan on the cluster counters for each task.

Triangle indices reordering Each triangle is processed by a single thread in parallel. The thread determines the nearest representative using the index stored in the previous phase. Then it determines the position of the triangle index by atomically incrementing the prefix scan value of the corresponding cluster counter taking into account also the parent node offset. The thread also assigns to the triangle the index of the corresponding task. If a triangle belongs to a leaf then the index is set to a special negative constant value. If the corresponding task was put to the leaf queue then we use bitwise negation to distinguish triangles belonging either to active or leaf tasks. At the end of the iteration input and output queues are swapped. This procedure is repeated until the output queue is empty.

Bounding boxes computation We compute bounding boxes of the k-ary BVH using a simple bottom-up refit procedure. If the current loop of the algorithm has been executed with $k = 2$ the loop is terminated and the postprocessing is applied.

Agglomerative clustering To transform k-ary BVH to binary BVH we build intermediate levels of the tree using naïve agglomerative clustering. This can be done in a single kernel launch. Each node of the k-ary BVH is processed by a single thread. We store treelet node indices in the local memory. In each step two nodes minimizing their unified surface area are merged together using a new parent node. The index of the first merged node is replaced by the parent node index and the index of the second merged node is replaced by the last node index. This preserves continuity of the indices array. This procedure is repeated until the whole treelet is constructed.

Leaf refinement The leaf queue consists of nodes which contain less than mk triangles (m is the maximum number of triangles per leaf). Splitting these into k clusters could create leaf nodes with very small number of triangles or even empty leaves. Therefore we postpone subdividing these nodes to an additional pass using $k = 2$. In this pass the task indices of triangles belonging to nodes in the leaf queue have to be activated. This concerns all triangles with negative task index except those with a special negative constant value used to mark triangles contained in the already finalized leaves. We use a simple kernel applying bitwise negation to negative task indices to get the original task index value for each such triangle. Then we swap the input queue and the leaf queue. We set k to 2 and we repeat the whole procedure again.

Postprocessing In post processing we run an additional kernel that copies the BVH into a new BVH with nodes allocated in the breadth-first order which is more efficient for ray tracing. Additionally, in this pass the empty leaves (if any) are removed from the BVH.

Note that we used the following optimization in our code: as the CUDA floating point atomic operations are rather slow we use fixed point coordinates and integer atomic add operation. We pre-multiply normalized coordinates by the maximum integer value divided by the number of triangles belonging to the corresponding node to ensure that the sums computed during the k-means loop do not overflow.

6 Results and Discussion

We have evaluated the proposed method using nine test scenes of different complexity. We used five different parameter settings that represent different goals in terms of the quality of the constructed BVH. The parameters of our method are: the number of clusters generated in one step of the k-means algorithm (k), the number of draws of initial representatives (p), and the number of iterations of the k-means algorithm (i). The selected five parameter sets are:

- k-means Q_1: $k = 8$, $p = 5$, $i = 0$,
- k-means Q_2: $k = 8$, $p = 5$, $i = 2$,
- k-means Q_3: $k = 16$, $p = 5$, $i = 5$,
- k-means Q_4: $k = 32$, $p = 20$, $i = 10$.
- k-means Q_5: $k = 64$, $p = 30$, $i = 15$.

As reference methods we used a full sweep SAH CPU builder implemented in C++ sequential code, the LBVH builder proposed by Karras [21], the HLBVH builder proposed by Garanzha et al. [13], and the ATRBVH builder proposed by Domingues [9]. For LBVH as well as HLBVH we used 60-bit Morton codes, HLBVH used 15 bits for the SAH based top-tree construction. For ATRBVH we used the publicly available implementation of treelet restructuring, using treelets of size 9 and 2 iterations.

We assume that the build time of SAH method is 0 which in turn gives us idealized time-to-image results using the full sweep SAH. In all cases the BVH termination criterion was set to 8 triangles per leaf. We evaluated the constructed BVH using a high performance ray tracing kernel of Aila et al. [2]. All measurements were performed on a PC equipped with Intel Core I7-3770 3.4 GHz, 16 GB RAM and GTX TITAN Black with 6 GB RAM.

The results are summarized in Table 1. For each method we report the SAH cost of the constructed BVH (using traversal and intersection constants $c_T = 3$ and $c_I = 2$), the average trace speed, the build time, and the time-to-image (total time) for two different application scenarios (the sum of kernel times is used). The first time-to-image measurement corresponds to path tracing with 8 samples per pixel, the second measurement corresponds to 128 samples per pixel, both use 1024x768 image resolution. Our path tracing implementation uses next event estimation with two light source samples per hit and Russian roulette for path termination. The reported times are an average of three different representative camera views to reduce the influence of view dependency.

BVH quality From Table 1 we can observe that the ATRBVH method provides the lowest BVH cost for all test scenes. However, regarding the trace speed the best performance was achieved by ATRBVH for 3 test scenes, while in other 6 cases the k-means based methods achieved the highest trace speed. While this can partly be caused by the view dependency of the measurements it also corresponds to recent observation of Aila et al. [1] that the SAH cost alone need not precisely reflect the trace speeds on the GPU.

Build time The results show that the LBVH method is the fastest in all test scenes. However the k-means Q_1 has build times relatively close to LBVH while providing better trace speed in 6 test scenes. The build times of our method significantly depend on the parameter settings. The k-means Q_1 is more than an order of magnitude faster than the k-means Q_5 method. Compared to ATRBVH, the k-means Q_1 is about 3x faster in all tests, while the ATRBVH build time is between the times of the k-means Q_2 and Q_3 methods. On smaller scenes k-means Q_1 is up to 4x faster than HLBVH, however for larger scenes (San Miguel, Power Plant) it becomes about 15-30% slower. This shows that the overhead per one input primitive is larger for our method than for HLBVH which uses global presorting of all primitives using Morton codes. However as our method is able to construct higher quality BVHs than HLBVH, the longer construction time can be amortized even in these more complex scenes.

Time-to-image Regarding the total time for lower number of path tracing rays the best times have been achieved by the LBVH (2 scenes), HLBVH (1 scene), ATRBVH (3 scenes), and k-means (3 scenes). Regarding the total time for higher number of rays the best times have been achieved by the ATRBVH (7 scenes) and k-means (2 scenes). These results indicate that the ATRBVH method seems to be the currently best choice for general usage, while the k-means based methods can provide slightly better performance for some scenes when setting up correct parameters. A visual comparison of the BVH cost and time-to-image for all tested scenes is given in Figure 4, where we show the results relative to the results of the idealized SAH method (full sweep SAH with 0 build time).

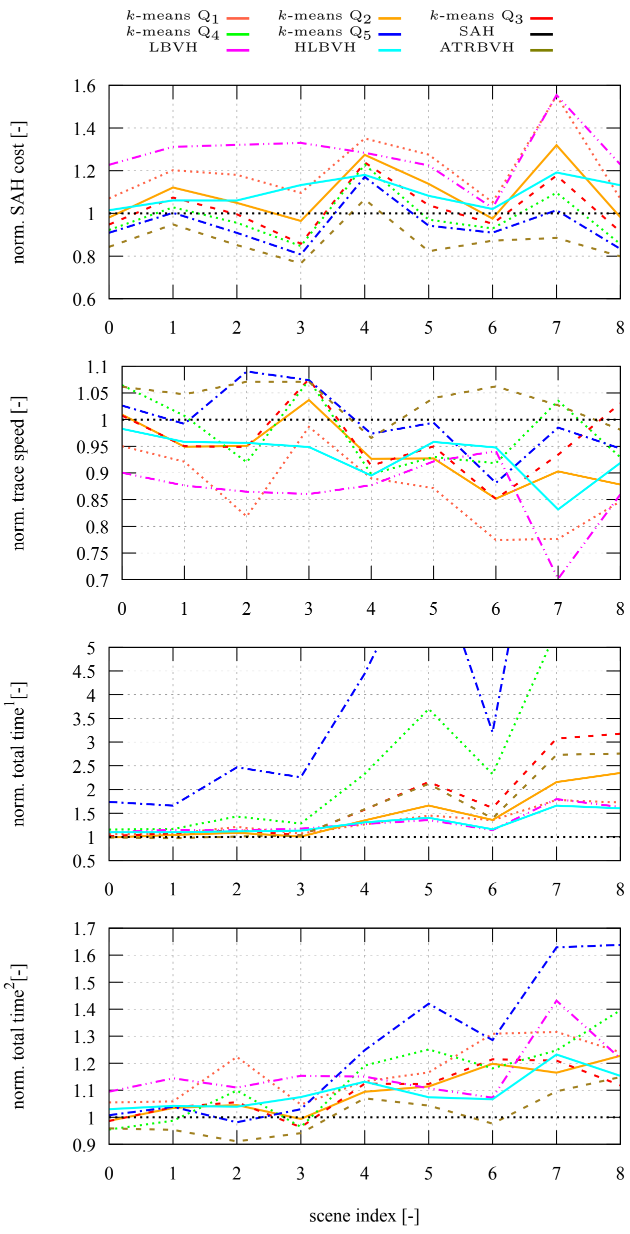


Fig. 4 Plots of the total time and SAH costs for all tested scenes. The depicted values are normalized with the respect to the idealized SAH method.

Method parameters We have extensively evaluated the dependence of the proposed method on its parameters by performing measurements for 88 parameter combinations. We used $k \in \{8, 16, 32, 64\}$, i in the range from 0 to 15, and p in the range from 5 to 30. From these measurements we selected 5 representative parameter settings that give a good overview of the behavior of the method (Q_1-Q_5 shown in Table 1). We have explicitly evaluated the dependence of the method on the number of k-means iterations used (see Figure 5). These results illustrate that particularly the first two k-means iterations are important and using more iterations does not really bring a benefit especially considering the corresponding linear increase in build times.

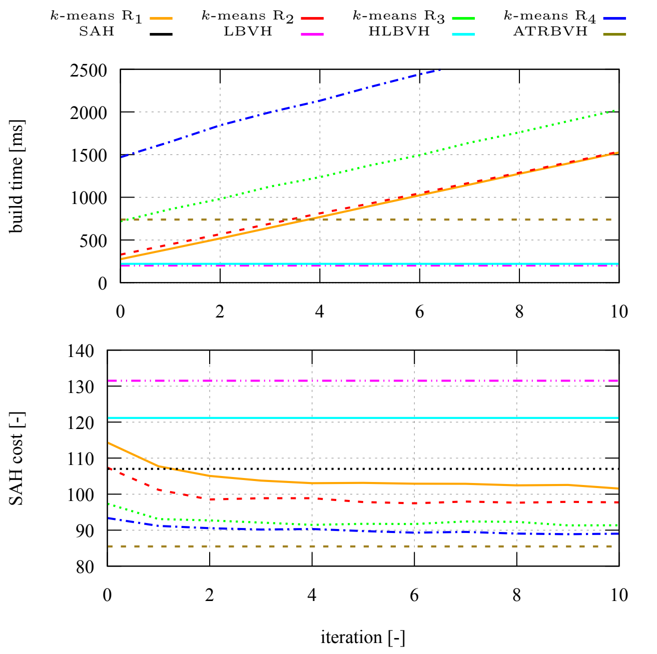


Fig. 5 Plots of the build time and SAH cost for the Power Plant scene showing the dependence on the number of iterations. We used four configurations: R_1 ($k = 8$, $p = 5$), R_2 ($k = 16, p = 5$), R_3 ($k = 32, p = 20$) and R_4 ($k = 64, p = 30$).

We have also measured the times required for different parts of our method using GPU timers (see Figure 6). We can observe that in all cases more than half of the build time is spent in the internal loop of the k-means algorithm in which the triangles are assigned to clusters and new cluster representatives are computed. Thus further optimizing this part of the algorithm might bring significant performance gains.

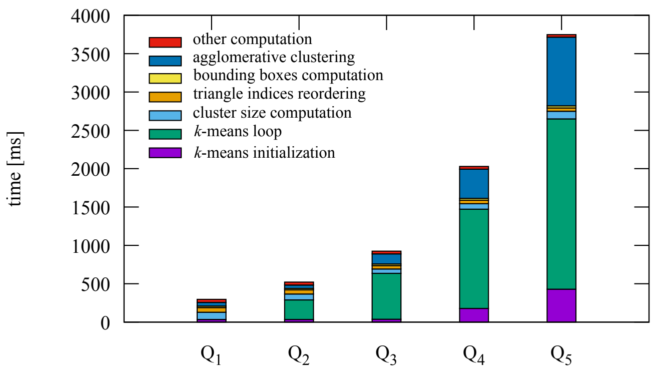


Fig. 6 Kernel times of different phases of the BVH construction for the Power Plant scene for the tested configurations.

Comparison with the AAC Although our method is primarily targeted at the GPU implementation we provide a brief comparison with respect to the state-of-the-art CPU BVH construction algorithm, namely the Approximate Agglomerative Clustering (AAC) method proposed by Gu et al. [15]. For the comparison we used the publicly available implementation of AAC provided by the authors of the method in the proposed Fast and HQ settings. This implementation is sequential and thus to get a more realistic picture about the performance of the method on a multi-core CPU we divided the AAC running times by the number of physical cores in the testing PC (i.e. by 4) to get a lower-bound of parallel running times.

The AAC-Fast is about 2-4 times slower than k-means Q_1 for the tested scenes (e.g. 956ms vs 274ms for the Power Plant scene), but leads to 10%-40% better SAH costs. The exception is the Power Plant scene for which the AAC implementation constructs BVHs with significantly worse cost than the k-means based methods (198 for AAC-Fast and 144 for AAC-HQ versus 115 for k-means Q_1 and 90 for k-means Q_5). The build times of AAC-HQ roughly correspond to build times of k-means Q_5, while the achieved SAH cost is 0.5%-10% better for the AAC (except for the Power Plant scene). These results indicate that our method is on pair with the state-of-the-art CPU builder and thus the choice of the method in practice should be motivated by the target platform and given application scenario.

7 Conclusion and Future Work

We proposed a new BVH construction method based on a combination of the top-down divisive clustering using the k-means algorithm and a bottom-up agglomerative clustering. The method uses several parameters which can be used to trade the quality of the constructed BVH for the construction speed. We described a parallel implementation of the method using CUDA. The results show that in a number of test cases the proposed method compares favorably to the HLBVH method. Compared to the more recent ATRBVH which represents a state of the art technique among GPU BVH builders our method is usually slightly worse than ATRBVH, but still leads to better results in several of the test cases. We expect that the k-means loop of our method can be further optimized and thus become even more competitive.

Since our method can directly build BVH with different branching factors, we believe that it can be used as a powerful tool opening new possibilities for the GPU based BVH construction. We also plan to further investigate other metrics for the k-means clustering algorithm, which might better reflect the axis aligned shape of the bounding volumes used for the constructed clusters. Further, we plan to investigate the possibility of incorporating the prediction of object movement in the clustering metric to construct a BVH optimized for several frames.

Acknowledgements

This research was supported by the Czech Science Foundation under research program P202/12/2413 (Opalis) and the Grant Agency of the Czech Technical University in Prague, grant No. SGS16/237/OHK3/3T/13.

References

1. Aila, T., Karras, T., Laine, S.: On Quality Metrics of Bounding Volume Hierarchies. In: In Proceedings of High Performance Graphics, pp. 101–108. ACM (2013)
2. Aila, T., Laine, S.: Understanding the Efficiency of Ray Traversal on GPUs. In: Proceedings of HPG, pp. 145–149 (2009)
3. Apetrei, C.: Fast and Simple Agglomerative LBVH Construction. In: R. Borgo, W. Tang (eds.) Computer Graphics and Visual Computing (CGVC). The Eurographics Association (2014). DOI 10.2312/cgvc.20141206
4. Arthur, D., Vassilvitskii, S.: K-means++: The Advantages of Careful Seeding. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '07, pp. 1027–1035. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2007)
5. Bittner, J., Hapala, M., Havran, V.: Fast Insertion-Based Optimization of Bounding Volume Hierarchies. Computer Graphics Forum **32**(1), 85–100 (2013)
6. Bittner, J., Havran, V.: RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In: Proceedings of SCCG, pp. 61–67. ACM (2009)
7. Dammertz, H., Hanika, J., Keller, A.: Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. Computer Graphics Forum **27**, 1225–1233(9) (2008)
8. Dasgupta, S.: The Hardness of k-means Clustering. Department of Computer Science and Engineering, University of California, San Diego (2008)
9. Domingues, L.R., Pedrini, H.: Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring. In: Proceedings of the 7th Conference on High-Performance Graphics, pp. 13–20 (2015)
10. Fabianowski, B., Fowler, C., Dingliana, J.: A Cost Metric for Scene-Interior Ray Origins. Eurographics, Short Papers pp. 49–52 (2009)
11. Feltman, N., Lee, M., Fatahalian, K.: SRDH: Specializing BVH Construction and Traversal Order Using Representative Shadow Ray Sets. In: Proceedings of HPG, pp. 49–55 (2012)
12. Ganestam, P., Barringer, R., Doggett, M., Akenine-Möller, T.: Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees. Journal of Computer Graphics Techniques (JCGT) **4**(3), 23–42 (2015)
13. Garanzha, K., Pantaleoni, J., McAllister, D.: Simpler and Faster HLBVH with Work Queues. In: Proceedings of HPG 2011, pp. 59–64. ACM SIGGRAPH/Eurographics, Vancouver, British Columbia, Canada (2011)
14. Goldsmith, J., Salmon, J.: Automatic Creation of Object Hierarchies for Ray Tracing. IEEE Computer Graphics and Applications **7**(5), 14–20 (1987)
15. Gu, Y., He, Y., Fatahalian, K., Blelloch, G.E.: Efficient BVH Construction via Approximate Agglomerative Clustering. In: Proceedings of High Performance Graphics, pp. 81–88. ACM (2013)
16. Havran, V., Herzog, R., Seidel, H.P.: On the Fast Construction of Spatial Data Structures for Ray Tracing. In: Proceedings of IEEE Symposium on Interactive Ray Tracing 2006, pp. 71–80 (2006)
17. Hunt, W.: Corrections to the Surface Area Metric with Respect to Mail-Boxing. In: Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on, pp. 77–80 (2008)
18. Hunt, W., Mark, W.R., Fussell, D.: Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In: Proceedings of Symposium on Interactive Ray Tracing, pp. 47–54 (2007)
19. Ize, T., Wald, I., Parker, S.G.: Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In: Proceedings of Symposium on Parallel Graphics and Visualization '07, pp. 101–108 (2007)
20. Jain, A.K., Dubes, R.C.: Algorithms for Clustering Data. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1988)
21. Karras, T.: Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In: Proceedings of High Performance Graphics, pp. 33–37 (2012)
22. Karras, T., Aila, T.: Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In: Proceedings of High Performance Graphics, pp. 89–100. ACM (2013)
23. Kay, T.L., Kajiya, J.T.: Ray Tracing Complex Scenes. In: D.C. Evans, R.J. Athay (eds.) SIGGRAPH '86 Proceedings), vol. 20, pp. 269–278 (1986)
24. Kensler, A.: Tree Rotations for Improving Bounding Volume Hierarchies. In: Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing, pp. 73–76 (2008)
25. Křivánek, M., Morávek, J.: *NP*-Hard Problems in Hierarchical-Tree Clustering. Acta Inf. **23**(3), 311–323 (1986)
26. Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., Manocha, D.: Fast BVH Construction on GPUs. Comput. Graph. Forum **28**(2), 375–384 (2009)
27. Lloyd, S.: Least Squares Quantization in PCM. IEEE Trans. Inf. Theor. **28**(2), 129–137 (1982)
28. MacQueen, J.: Some Methods for Classification and Analysis of Multivariate Observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics, pp. 281–297. University of California Press, Berkeley, Calif. (1967)
29. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. Queue **6**(2), 40–53 (2008)
30. Pantaleoni, J., Luebke, D.: HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In: Proceedings of High Performance Graphics '10, pp. 87–95 (2010)
31. Rubin, S.M., Whitted, T.: A 3-Dimensional Representation for Fast Rendering of Complex Scenes. In: SIGGRAPH '80 Proceedings, vol. 14, pp. 110–116 (1980)
32. Vinkler, M., Bittner, J., Havran, V., Hapala, M.: Massively Parallel Hierarchical Scene Processing with Applications in Rendering. Computer Graphics Forum **32**(8), 13–25 (2013)

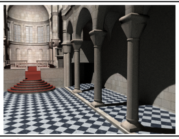
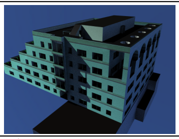
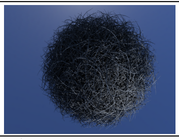
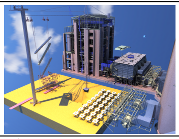
			Sponza #triangles 66k					Sibenik #triangles 75k					Crytek Sponza #triangles 262k		
	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time^1 [ms]	total time^2 [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time^1 [ms]	total time^2 [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time^1 [ms]	total time^2 [ms]
SAH	216	211	-	141	2225	147	199	-	176	2792	213	162	-	166	2660
LBVH	264	189	**1.8**	154	2429	192	175	**2.1**	202	3197	279	140	**4.8**	189	2944
HLBVH	219	206	11.7	154	2292	156	192	12.3	193	2911	225	155	14.1	186	2764
ATRBVH	**180**	223	5.4	**139**	2130	**138**	**209**	6.0	**171**	**2663**	**180**	175	16.6	**167**	**2418**
k-means Q$_1$	229	200	2.7	148	2341	177	184	3.0	187	2957	251	133	5.7	201	3248
k-means Q$_2$	211	212	4.2	**139**	2196	165	190	4.5	183	2888	222	154	10.2	180	2776
k-means Q$_3$	203	212	9.0	143	2191	158	190	9.3	189	2895	211	154	17.8	190	2805
k-means Q$_4$	198	**224**	30.1	162	**2121**	150	201	33.7	205	2756	203	150	60.2	237	2920
k-means Q$_5$	195	216	111	244	2238	147	199	117	291	2899	192	**177**	260	408	2604
			Conference #triangles 331k					Happy Buddha #triangles 1087k					Soda Hall #triangles 2169k		
	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time^1 [ms]	total time^2 [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time^1 [ms]	total time^2 [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time^1 [ms]	total time^2 [ms]
SAH	120	223	-	214	3425	168	109	-	118	1834	219	205	-	108	1644
LBVH	159	192	**5.8**	251	3947	216	96	**18.9**	151	2111	270	189	**34.2**	**146**	1821
HLBVH	135	212	13.5	242	3679	198	97	26.4	155	2076	237	197	43.2	151	1766
ATRBVH	**93**	239	20.4	221	**3223**	**180**	105	67.1	187	**1961**	**180**	**213**	129	228	**1716**
k-means Q$_1$	131	220	6.8	230	3586	227	97	21.7	**150**	2080	280	180	39.0	156	1920
k-means Q$_2$	115	231	12.3	**215**	3403	213	102	36.7	160	2008	250	190	69.2	179	1832
k-means Q$_3$	102	**240**	22.4	222	3299	207	100	61.3	186	2062	228	195	125	232	1847
k-means Q$_4$	101	**240**	75.0	275	3302	207	98	147	275	2184	213	191	285	397	2056
k-means Q$_5$	96	**240**	282	484	3525	195	**106**	406	524	2285	207	203	665	770	2333
			Hairball #triangles 2880k					San Miguel #triangles 7880k					Power Plant #triangles 12759k		
	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time^1 [ms]	total time^2 [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time^1 [ms]	total time^2 [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time^1 [ms]	total time^2 [ms]
SAH	1257	48	-	329	4962	171	116	-	267	4056	108	70	-	420	6605
LBVH	1290	45	**46.1**	**376**	5322	264	81	**129**	479	5803	132	61	**200**	682	8043
HLBVH	1284	45	54.6	381	5291	204	96	142	**443**	4993	120	65	214	**670**	7611
ATRBVH	**1098**	**51**	165	456	**4844**	**150**	119	483	728	**4439**	**84**	69	731	1151	7575
k-means Q$_1$	1324	38	52.7	444	6501	264	90	163	473	5333	115	60	274	721	8195
k-means Q$_2$	1227	42	90.0	447	5946	226	104	302	576	4720	105	62	518	981	8132
k-means Q$_3$	1197	41	169	529	6029	201	108	551	820	4905	98	**72**	927	1326	**7382**
k-means Q$_4$	1170	44	427	762	5860	187	**120**	1189	1429	5058	92	67	2010	2467	9406
k-means Q$_5$	1144	42	707	1052	6381	173	114	2522	2778	6602	90	66	3782	4203	10786

Table 1 Performance comparison of tested methods. We used five configurations: Q$_1$ ($k = 8$, $p = 5$, $i = 0$), Q$_2$ ($k = 8$, $p = 5$, $i = 2$), Q$_3$ ($k = 16$, $p = 5$, $i = 5$), Q$_4$ ($k = 32$, $p = 20$, $i = 10$) and Q$_5$ ($k = 64$, $p = 30$, $i = 15$). The reported numbers are averaged over three different viewpoints for each scene. The best results are highlighted in bold. For computing the SAH cost we used $c_T = 3$ and $c_I = 2$.

33. Wald, I.: On fast Construction of SAH based Bounding Volume Hierarchies. In: Proceedings of the Symposium on Interactive Ray Tracing, pp. 33–40 (2007)
34. Wald, I.: Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. IEEE Transactions on Visualization and Computer Graphics **18**(1), 47–57 (2012)
35. Walter, B., Bala, K., Kulkarni, M., Pingali, K.: Fast Agglomerative Clustering for Rendering. In: IEEE Symposium on Interactive Ray Tracing, pp. 81–86 (2008)
36. Weghorst, H., Hooper, G., Greenberg, D.P.: Improved Computational Methods for Ray Tracing. ACM Transactions on Graphics **3**(1), 52–69 (1984)