gkDtree : A Group-Based Parallel Update Kd-tree for Interactive Ray Tracing

Yoon-Sig Kang[a], Jae-Ho Nah[a], Woo-Chan Park[b], Sung-Bong Yang[a]

[a]*Dept. of Computer Science, Yonsei University, Korea*
[b]*Dept. of Computer Engineering, Sejong University, Korea*

Abstract

This paper proposes a new group-based acceleration data structure called gkDtree for interactive ray tracing of dynamic scenes. The main idea of the gkDtree is to construct the acceleration structure with a multi-level hierarchy, and to integrate a parallelization approach to result in a faster update and a more efficient tree traversal. A gkDtree can be viewed as a set of kd-trees, each of which is a local acceleration structure corresponding to a group. For a gkDtree, a scene is divided into several groups based on a scene graph. The local acceleration structure of each group involving only dynamic primitives is rebuilt. To achieve higher parallelization, dependencies among groups in different levels are removed before rebuilding occurs in parallel. To enhance the scalability of parallelization, a simple and fast load-balancing scheme is introduced. Furthermore, we applied a variety of accurate SAH (surface area heuristic) algorithms into tree generation for both static and dynamic groups. The experimental results show that a gkDtree has a real-time update performance. It has an update performance that is up to 166 times faster than a kd-tree for our test scenes in an six-core hardware system environment. Furthermore, the results also show that tree traversal performance of a gkDtree is competitive with that of a kd-tree.

1. Introduction

Ray tracing is a rendering technique that generates photo-realistic images by tracing the photons that emanate from light sources. However, ray tracing requires enormous computations for simulating the reflections and refractions of the photons in 3D scenes. Hence it has mainly been used in off-line rendering. Recently, real-time ray tracing has received close attention from researchers owing to faster hardware and algorithmic improvements.

Ray tracing can be accelerated by the use of data structures such as kd-trees, octrees, grids, and bounding volume hierarchies (BVHs). An acceleration structure enhances the rendering performance by reducing the number of intersection tests between rays and primitives being rendered.

Since the primitives in dynamic scenes may move their positions from frame to frame, an acceleration structure should be updated every frame. Therefore, building and updating an acceleration structure greatly influences the overall performance of an interactive ray tracer for dynamic scenes. Due to this fact, various acceleration structures have been extensively studied. Among a wide variety of interactive ray tracing approaches [1], the multi-level hierarchy and parallelization approaches are of most concern in this paper, since they are used for our acceleration structure.

In the multi-level hierarchy approach, primitives in a specific scene are divided into multiple groups, each of which has the same linear transformation [2]. This method provides a fast update for the scenes by updating only those groups that contain moving objects and inversely transforming the rays rather than the objects. However, the update performance cannot be improved if deformable primitives exist in the scenes. In addition, the rendering performance deteriorates due to the inverse transformation of the rays [1].

Furthermore, there have been studies of parallelization approaches to process the building and updating of acceleration structures on multi-core CPUs [3, 4, 5] and GPUs [6, 7, 8, 9]. These approaches obtain enough independent threads by dividing the entire tree into subtrees. For this upper level tree construction, an exact surface area heuristic (SAH) [5] or alternative subdivision heuristics [3, 4, 6, 7] are used. The first provides no tree quality degradation, but spends an inordinate amount of time on tree construction. The latter is performed more quickly, but causes tree quality degradation.

In this paper, we propose a group-based acceleration data structure, called gkDtree. The proposed data structure is a set of kd-trees that incorporate both the multi-level hierarchy and parallelization approaches. For a gkDtree, a scene is partitioned hierarchically using its scene graph. Only the local acceleration structure of a group that contains dynamic primitives is rebuilt. Each group in a gkDtree is organized hierarchically as well. Therefore, with the exception of the leaf groups in a group hierarchy, split candidates decrease from primitive boundaries to group node boundaries. This is similar to [10]. This group-based SAH evaluation is faster than the exact SAH evaluation [5, 11, 12].

In the multi-level hierarchy, it can be difficult to achieve a high level of

parallelization during the rebuilding of the tree. The reason is that the nodes in the higher level of the tree can be updated after the nodes in the lower level of the tree have been updated. To resolve this problem, a gkDtree performs the geometry update stage before the rebuilding of the local acceleration structure of a group. In the geometry update stage, the axis-aligned bounding boxes (AABBs) of the animated primitives and groups are only updated in a bottom-up fashion without rebuilding the local acceleration structure. Such updates eliminate the dependencies among the nodes in the tree. Furthermore, a gkDtree performs load-balancing for efficient parallelization. This scheme distributes the groups to threads, level by level, from the bottom-up in a round-robin fashion. This scene graph-based parallelization approach provides higher tree quality than the object median-based parallelization [3].

Through the aforementioned schemes, a gkDtree exhibits an update performance that is up to 166 times faster update performance than a kd-tree [13] for the BART test scenes [14] and the Conference scene in a six-core hardware system environment (Figure 1). Even comparing with [3] and [8], the update performance of a gkDtree is higher on the BART test scenes than that of [3] and [8]. Furthermore, the experimental results also indicate that the tree traversal performance of a gkDtree is competitive with that of a kd-tree. We can obtain these results by applying different SAH algorithms to tree generation for both static and dynamic groups. We use three axes for exact SAH calculations with small leaf sizes in static groups, while we use the longest axis for fast SAH calculations with small large sizes in dynamic groups.

The remainder of this paper is organized as follows. Section 2 describes the related work regarding this study and Section 3 explains the detailed concepts of a gkDtree. Section 4 illustrates the results of our experiments. Finally, in Section 5 conclusions are drawn.

Figure 1: The scenes rendered by the proposed acceleration structure: BART-Kitchen, BART-Robot, BART-Museum, and Conference.

2. Related Work

There have been great efforts to increase the speed of interactive ray tracing for dynamic scenes. In this section, we review scene update, parallel update, and tree construction strategies.

2.1. Scene Update Strategy

Acceleration structures should be adjusted for moving objects in dynamic scenes. They can be rebuilt from scratch for each frame [3, 4, 15, 11]. In this case, precomputed triangle clusters can be used to reduce the input size as shown in [16]. Other researchers [17, 18, 19] afford partial updates for the portions that require changes due to animation in successive frames. And lastly, if the scenes are almost static, an acceleration structure for the static geometry can be constructed before rendering and is reused for successive frames [2, 20].

The rebuild approach generates high quality data structures and exhibits excellent efficiency when a large number of intersection tests are performed and the size of the scene data is relatively small. However, the static update approach is only able to show higher tree update performance for the scenes that do not contain any deformable primitives. The update approach, in contrast, can be used in the scenes containing deformable primitives, providing better tree update performance than the rebuild approach in general.

There are several disadvantages to these approaches. First, the rebuild approach proves to have inferior rebuild (or update) performance, since it must rebuild the structure for each frame in the entire scene. For partial rebuild, Arauna [21] divided the entire tree into static parts and dynamic parts. However, this increases the number of traversal steps, since it always traverses two divided trees. Second, the static approach undergoes performance degradation in the intersection tests. Its overall performance can deteriorate more rapidly as the number of rays increases, for generating high quality images. Moreover, it suffers from further performance degradation if the scenes contain a large number of deformable primitives. Third, in the update approach, the quality of the data structure is continuously degraded as updates are performed, consequently resulting in slower intersection test performance. To relieve such degradation, there are studies [22, 23] which have combined the update and rebuild approaches.

2.2. Parallel Update

On multi-core CPUs, previous researchers have proposed the parallel construction of grids [24], kd-trees [3, 5, 25, 26], and BVHs [4], respectively. Others have exploited graphics processing units (GPUs) for the parallel construction of grids [27, 28], kd-trees [6, 8], and BVHs [7, 9], respectively. We provide a brief overview of these parallel methods in this section.

First of all, we will introduce CPU tree build methods. The SAH [12], used in tree building in general, is usually recursively performed in a top-down fashion. Thus, early parallel tree build algorithms have shown lower scalability [25, 26]. In contrast, [3] has developed a highly parallel kd-tree construction algorithm that overcomes the limits shown in [25, 26]. This algorithm uses the object median to divide a tree into subtrees in order to achieve scalability. However, this decomposition may impair the rendering performance due to tree quality degradation. Recently, [5] presented a novel parallel decomposition method based on the exact SAH [11]. Although this method constructs the same tree as [11], its tree build performance is not superior to [3]. In the case of BVHs, two methods for parallel BVH construction were first presented by [4]. One method is a combination of vertical parallelization using partitioned subtrees and horizontal parallelization using multiple threads working in the same binning step. The other method uses partitioned sub-domains made by uniform grids.

Next, GPU tree construction methods were introduced recently. [6] proposed a kd-tree build algorithm using programmable shaders in GPU. This algorithm requires a considerably larger amount of memory than CPU-based approaches, since it builds trees in the breath-first order to increase the scalability of shaders. The GPU memory size also limits tree depth. To relieve this problem, [8] proposed a partial breadth-first search construction, and reduced the size of the working set. In the case of BVHs, [7] proposed a hybrid BVH construction algorithm on GPU, which incorporates LBVH (linear bounding volume hierarchy) construction using Morton codes and an SAH-based BVH build.

Finally, parallel grid construction has been studied for ray tracing of dynamic scenes. Various parallel methods for construction of uniform grids were studied by [24], and scalability was measured on a 16-core machine. GPU parallelization of uniform grid construction and two-level grid construction were presented by [27] and [28], respectively. Although these methods provide fast update performance, the grids are less efficient for traversal and intersection tests than tree structures [1].

2.3. Tree Construction Strategy

The SAH [12, 29], has been widely used for tree construction. Although the SAH provides high quality trees, the original SAH requires O(nlog^{2}n) computation costs. [11] presented an O(nlogn) kd-tree construction method using initial sorting and sort free sweeping.

Several researchers [3, 15, 26] have applied O(n) binned radix sort to SAH approximation. This method greatly reduces the SAH computation cost, but it makes rendering performance degradation due to the selection of split planes by sampling.

Finally, hierarchies also have been utilized to reduce the split plane candidates. The Razor system [10, 30] constructs an object-based, multi-level hierarchy, using scene graph hierarchies and performing lazy updates in local acceleration structures. Hierarchical linear bounding volume hierarchy (HLBVH) [9], an improved version of the original LBVH [7], has presented a node hierarchy emission procedure. This method generates treelet hierarchies from sorted Morton codes to reduce global memory traffic on GPUs.

3. gkDtree

In this section, we describe details of the gkDtree. First, the construction method of the initial gkDtree is explained. We then show how to update gkDtree according to the animation. Finally, the parallel scheme for building local acceleration structures of groups and the gkDtree traversal method are provided.

3.1. Initial gkDtree Construction

The construction of the gkDtree consists of two steps. In the first step, a group hierarchy is generated from a hierarchical animation data structure (e.g., a scene graph). Then, a local acceleration structure for each node in the group hierarchy is built.

Before describing the process of group hierarchy construction, we need to define a group. In a gkDtree, a group is treated as a primitive. A group has the following member variables. First, like a kd-tree, a group has three pointers to the node data, the primitive list, and the primitive data. However, there is a difference between a gkDtree and a kd-tree. In a gkDtree, a group's pointers reference its local data, while the pointers in kd-tree indicate global data. In a gkDtree, this local data is used for the construction of the local acceleration structure. Second, a group has the update flag that indicates whether the

group is static or dynamic. Third, a group has transform matrices. If a group is dynamic, these matrices are updated for each frame. Fourth, a group should know which level it is on within the group hierarchy. Finally, a group contains the AABB that includes the AABBs of the group's primitives.

The process of group hierarchy construction is simple, because we can easily construct a group hierarchy from a scene graph. In other words, a group hierarchy is basically similar to a scene graph hierarchy. Therefore, to construct a group hierarchy, we should recursively traverse a scene graph. Nevertheless, there are a couple of differences between a group hierarchy and a scene graph hierarchy. First, if a scene graph node has only one child, the parent scene graph node and the child node are merged into a group node for faster traversal. This case occurs when multiple transforms (e.g., translation, rotation, scaling, and matrix operations) are performed consecutively. If we don't apply this scheme, redundant ray-AABB intersection tests are performed, since the parent and its child group node both have the same AABBs in this case. Second, if a scene graph node is static and it is at a level 1 (next to the root), we create a group node. That is, even though static objects consist of multi-level hierarchies, we only construct a two-level hierarchy. Because static objects are not updated after the first frame, multi-level hierarchies are not necessary.

After a group hierarchy has been created, each local acceleration structure is constructed as a kd-tree with SAH. In general, kd-trees have been known to provide optimal rendering efficiency [31]. For local acceleration structures in a gKDtree, various data structures and algorithms (e.g., BVHs, grids, or binning algorithms) can be employed, instead of SAH-based kd-trees.

The process of local kd-tree construction is similar to the general kd-tree construction algorithm except for some optimization techniques. First, if a local kd-tree includes group nodes (the case of the upper-level local kd-tree), a large leaf size may degrade the overall performance due to excessive traversal of local kd-trees. Also, if a group nodes includes a few primitives (e.g., less than 16), a local kd-tree would be flattened. Thus, a smaller leaf size is suitable for these two cases. Second, we can use different tree build strategies to static and dynamic local trees, similar to [20]. Because static local trees are not updated after the first frame, the tree construction time in preprocessing is less important than the tree update time in the case of dynamic scenes. Therefore, more accurate SAH and a small leaf size can be used for static trees, even though these two settings require longer construction time. On the other hand, less accurate SAH and a large leaf size can be used for dynamic trees in order to accelerate tree

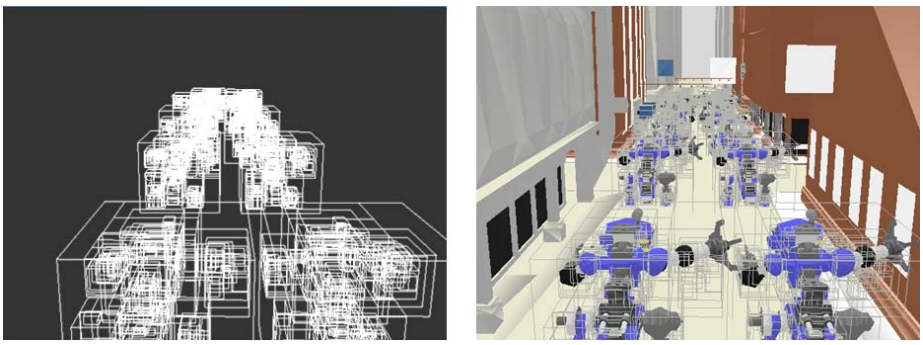

Figure 2: Wire framed gkDtree object hierarchy for the BART Robot scene. In higher level local kd-trees, these multi-level object boundaries are used for SAH calculation instead of actual primtives.

update. Details of our tree construction parameters will be described in Section 4.

Primitive sorting and SAH calculations in a gkDtree are more rapidly performed than those in kd-tree, because inner groups in a group hierarchy have group primitives instead of actual primitives. Therefore, the input data for tree construction is greatly reduced in this case. Figure 2 depicts the bounding boxes of the groups in the group hierarchy of the BART Robot. A gkDtree uses the boundaries of these bounding boxes instead of primitive's boundaries for the SAH calculation.

Figure 3 depicts a sample gkDtree. The root node of each local acceleration structure is referred to as a group node. A box in the tree indicates a group node, which includes transform information (Tn in Figure 3). A parent and its child nodes for consecutive transforms are merged into a group node, as mentioned earlier (T3 and T4 in Figure 3(b)). A box in the tree indicates a group node and a white circle is a leaf node whose children are either group nodes or primitives. A dark circle is an inner node that is inserted during the space partitioning. Observe that each shaded area indicates a local acceleration structure. Leaf nodes can include either group nodes or actual primitives. We regard such groups in the leaf nodes as primitives for the rest of the gkDtree processing.

3.2. gkDtree Update

The gkDtree update process consists of two stages: the geometry update and the tree update stages. In the geometry update stage, the transformation for

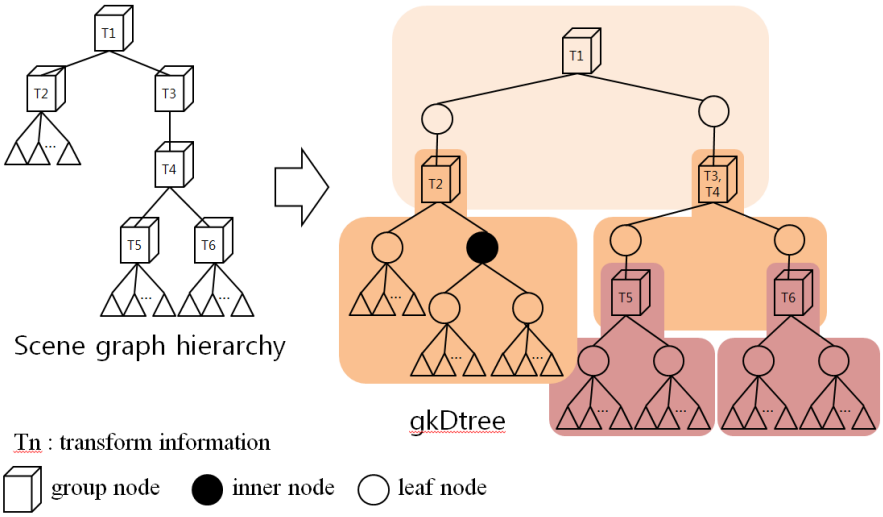

Figure 3: (a) A scene graph hierarchy, (b) A sample gkDtree with five different group nodes converted from the scene graph hierarchy.

each group that involves animation is performed. If any updates are conducted on a group, all primitives in this group should be updated as well. The advantage of this type of update is that the quality of the data structure becomes relatively high.

The primitive's AABBs that are a group's children should be compared to obtain a new AABB of the group. This implies that there exist dependencies among the groups in different levels of the tree. For parallel rebuild, we calculate the AABBs of all groups in a gkDtree, level by level, from the bottom up. After this calculation, each group in the tree has a new AABB. Hence, there is no more dependency among the groups in different levels. In the tree update stage, each local acceleration structure that requires update is completely rebuilt. We apply the general kd-tree construction method using the SAH for the rebuild. In the following subsection, we describe how tree updates can be done in parallel.

3.3. Parallelization

Both geometry and tree update stages of the gkDtree update can be processed in parallel. Heavy computations are involved in these stages. For the geometry update stage, group nodes in a gkDtree are assigned to the threads, level by level, from the bottom up, in a round-robin fashion (as shown in Figure 4). We assume that there are four threads available.

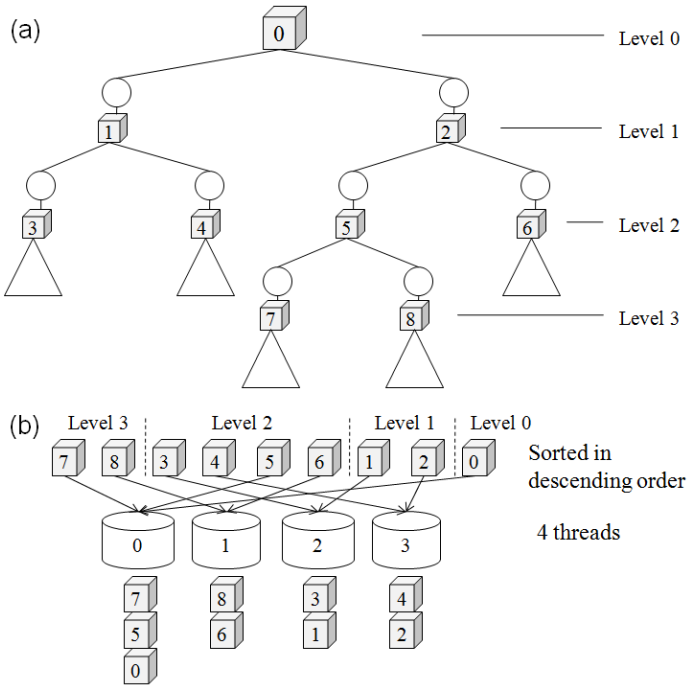

Figure 4: (a) a sample gkDtree, (b) allocation of groups in Figure 4-(a) to four threads for acceleration structure rebuild.

Once all of the dependencies among the groups are removed after the geometry update stage is done, all of the threads can process the groups allocated to them in parallel. It is obvious that we have to consider a reasonable load balancing among the threads in order to achieve higher parallelization. In order to assign the nodes evenly to the thread, it is helpful to know the amount of primitives associated with each node. However, too much time is necessary for a group node in an upper level of the tree to determine its number of primitives. That is, all of the nodes in the subtree that are rooted at the group node should be updated. Such load balancing could provide excellent results, but load balancing itself may become a bottleneck. Hence, we introduce a very simple, yet effective, load balancing scheme.

Before applying the load-balancing scheme, we should prepare an array of pointers to the groups, as shown in Figure 4-(b). An array of the pointers to the groups 7, 8, 3, 4, 5, 6, 1, 2, and 0, was given. Note that the group IDs are given in the level order traversal of the tree, beginning from the root, during the initial tree construction. The pointers in the array are ordered from the

bottom-most level, and from left to right on the same level. Our load-balancing scheme simply assigns groups to the available threads, one by one, according the order of the groups in the array. Figure 4 illustrates the scheme when there are four threads for a sample gkDtree with nine nodes. For example, group 7 is assigned to thread 0, group 8 to thread 1, group 3 to thread 2, group 4 to thread 3, group 5 to thread 0.

Although such assignment may not partition the computational loads into equal amounts for the threads, it works quite well since the groups on the same level have approximately similar amounts of computational loads. According to our experiment, for the BART Robot scene that contains a significantly large number of groups, this scheme enhances the update performance up to 54% (22.9ms $\rightarrow$ 14.8 ms). We applied this load-balancing scheme to further experiments in Section 4.

3.4. gkDtree Traversal

The tree traversal method of the gkDtree is quite similar to that of a general kd-tree. One major difference is that upon arrival at a group node, the traversal continues from the root of the local kd-tree of the group (refer to Algorithm 1 for more details). First, InnerNodeTraversal() in the gkDtree traversal is exactly the same as that in kd-tree traversal. This function finds the next-visiting nodes. If the ray visits both children nodes, the farer node, with respect to the ray, is pushed into the stack. Also, RayPrimitiveIntersectionTest() is the same as that in the general intersection test algorithms. However, LeafNodeTraversal() in the gkDtree differs from that in kd-tree traversal. In the leaf node traversal, a gkDtree recursively calls RayTrace() when a primitive's type is a group node. When these recursions of local group traversal are completely finished, we can determine the final hit point of the ray.

Because gkDtree traversal is similar to kd-tree traversal, gkDtree traversal may employ existing optimization methods for kd-trees like [20], such as mailboxing [32, 33] and masked packet traversal [34]. Applying mailboxing to gkDtrees is described in Algorithm 1. We assumed that gkDtrees use thread-local mailboxing algorithms such as inverse mailboxing in [33] because the original mailboxing algorithm in [32] can be harmful on multithreading environments as described in [33]. Because of the limited size of a ring buffer for inverse mailboxing, efficient mailboxing strategies for gkDtrees are needed.

We here present a simple tip for a gkDtree with mailboxing. Traversal of the same local tree multiple times is much more expensive than intersection tests

Algorithm 1: *gkDtree traversal*

```
function RayTrace(group, ray, hitinfo)
1.  if (TestAABB(group->min, group->max, ray, tmin, tmax) == true) then
2.   node = group->root;
3.   while (node != NULL and ray.maxt >= tmin)
4.     if (node->isLeaf() == true) then
5.       LeafNodeTraversal(node, ray, hitinfo);
6.       if (stack.isEmpty() == false) then
7.        stack.pop(node, tmin, tmax);
8.       endif
9.     else
10.       InnerNodeTraversal(node, tmin, tmax, stack);
11.    endif
12.  end while
13.  return true;
14. endif
15. return false;

function LeafNodeTraversal(node, ray, hitinfo)
1.  for (i=0; i<node->nprims ; i++)
2.    p = primitive[node->primList[i]];
3.    if (p->type == Group) then
4.      if (Mailboxing(groupMailbox, p)==true) then
5.        if (RayTrace(p, ray, hitinfo) == true) then
6.          Updatemailbox(groupMailbox, p);
7.        endif
8.      endif
9.    else
10.     if (Mailboxing(primMailbox, p)==true) then
11.       RayPrimitiveIntersectionTest(ray, hitinfo);
12.       Updatemailbox(primMailbox, p);
13.   endif
14.  endif
15. end for
```

with the same primitive multiple times. Therefore, whenever a ray traverses a local tree, the result of this traversal should be maintained in the mailbox data as far as possible in order to avoid unnecessary traversal. To efficiently manage this feature, we first divided a ring buffer into two buffers - one for group primitives and one for actual primitives. Although it requires double memory space, the additional space is still quite small. For example, if we use two 8-entry ring buffers, the additional memory space required is only 64 bytes per thread. Also, the cost of mailboxing is the same as that of a single ring buffer because we choose either of these two ring buffers according to primitive types. Next, the mailbox data is updated only if a ray intersects a group node's AABB. If a ray visits a group node again and the ray does not intersect with the group node, only a duplicated ray-AABB intersection test is performed. In contrast, if a ray visits a group node again and the ray intersects with the group node, duplicated tree traversal should be performed. Using our mailbox writing method, tree traversal of the latter case can be avoided more efficiently, as the former case does not pollute the mailbox data. In contrast to these techniques for group primitives, mailboxing for actual primitives is the same as the original inverse mailboxing.

Like other object-based, multi-level hierarchy acceleration structures, during the construction of the gkDtree, the empty spaces (the spaces that do not contain any primitive) are created without any special efforts, since splitting a node in a gkDtree is done with respect to the boundaries of the AABBs. In other words, the split plane becomes one of the AABB boundaries of the child group nodes. Therefore, when there is an empty space between two child nodes, the split plane is highly likely to be determined in such a way that the empty space will be maximized. Such an empty space is expected to enhance rendering efficiency [25].

4. Implementation and Experiment

We analyzed the efficiency of a gkDtree in comparison with a kd-tree. In order to ensure objectivity in the experiments, the common sections of both a kd-tree and a gkDtree were implemented to ensure equal performance. Furthermore, we used the same tree build parameters for building the acceleration structures on the three different algorithms. These parameters were the same as those in [13]: the intersection cost is 80, the traversal cost was 1, the empty bonus was 0.5, and the maximum tree depth was $8 + 1.3 \log_2 n$ (n: the number

of primitives). We also performed the experiments without the empty bonus because this empty bonus may affect tree quality. In the case of local kd-trees including groups, the empty bonus was not used because empty space could be naturally maximized in upper-level local trees as described in Section 3.4. Next, we computed the cost function only with respect to the longest axis to find the split planes as described in [13]. Finally, the maximum number of primitives in each leaf node was set to 8 for both kd-trees and gkDtrees. As described in Section 3.1, gkDtrees have two exceptions for the leaf size. A leaf size of 4 was used for local kd-trees including group nodes or a few primitives (less than 16). This configuration was helpful for avoiding excessive traversal as well as intersection tests.

Since the static groups in a scene do not need to be updated during rendering, their initial acceleration structures are maintained without modification. Hence, we can obtain higher tree traversal performance when we build the acceleration structures of static groups with more accurate cost functions in the preprocessing stage as described in [20]. In contrast, the acceleration structures of dynamic groups should be updated in every frame. Therefore, faster update is also important. To exploit this feature, we applied different SAHs to static and dynamic groups using split axis and leaf size. First, during the preprocessing stage for generating a gkDtree for the first frame, the acceleration structures for the static groups were constructed by selecting the split planes, based on the computations of the cost function with respect to all three axes (x, y, and z). Also, a leaf size of 4 was used for these static groups so that deeper trees could be created. Second, dynamic groups including actual primitives (in the case of lower-level local kd-trees) used a leaf size of 12 for faster update. To measure the effectiveness of this method, we have performed the experiments both with and without this approach.

The codes in [13] for the tree build and tree traversal algorithm (predominantly affecting the overall performance) were included in both codes with no modifications, with the exception of the portions for both multi-threading, inverse mailboxing [33], and packet tracing [34].

For the memory management of a gkDtree, one bit is needed to distinguish between a group node and a primitive. Generally, the number of group nodes is smaller than that of primitives as described in Table 1. In addition, group nodes also perform a role of inner nodes using its AABB values.

We tested the BART (Kitchen, Robot, and Museum) scenes for dynamic ray tracing performance in Table 1. Each of the three scenes shows a different

Table 1: Static and dynamic parts in the test scenes.

	Static parts		Dynamic parts		Total number of primitives
	Groups	Primitives	Groups	Primitives	
Kitchen	15	103K	94	7K	110K
Robot	1	9K	1911	62K	71K
Museum4	4	9K	1	1K	10K
Museum8	4	9K	1	65K	75K
Conference	278	252K			252K

type of animation. In the Kitchen scene, there are only about 7% of dynamic primitives in the entire scene. Therefore, a partial rebuild algorithm should work quite well on the Kitchen. The Robot scene, however, has about 87% of dynamic primitives. The dynamic primitives in the Robot are composed of a huge number of animation groups so that a large number of groups are generated for a gkDtree. These groups can be processed in parallel for a gkDtree (as described in Section 3) to achieve a higher level of scalability. Next, the Museum scene contains deformable primitives that are absent in other scenes, and allows the performance of tree rebuild/update algorithms to be measured. We have chosen two different settings: the Museum4 scene with 1K dynamic triangles and the Museum8 scene with 65K. In the former scene, gkDtree could utilize advantages in partial update because it has a few dynamic primitives. However, in the latter scene, the advantages in partial update diminish because there are a large portion of dynamic triangles in the scene. Finally, we have tested the Conference scene to measure the pure build/traversal performance of gkDtrees. Because this scene is static, we cannot use partial update and different SAH schemes for static and dynamic groups. Table 1 shows the composition of the five scenes (the numbers of groups and primitives for static and dynamic parts in each scene).

For the experiments, an Intel Core i7 980X (six core, 3.33 GHz, 12 Mbyte cache) with 6 Gbyte of memory was used. The number of threads varies from 1 to 12 for parallelization. If the number of threads was higher than six, hyper-threading was enabled. All images were rendered at 1024×1024 resolution with primary rays. We utilized single ray tracing and 2×2 packet tracing with Intel SSE (streaming SIMD extensions). We used a well-known ray-triangle intersection algorithm in [35]. Each value in Tables 2 and 4 is the average processing performance in ms per frame and frames per second (FPS), respectively. These values were obtained by calculating the averages of tree update and rendering

Table 2: Average acceleration structure update performance (in ms per frame) for the test scenes. In the Conference scene, the initial acceleration structure construction time is reported. S/D SAHs means applying different SAHs to static and dynamic groups. EB is an abbreviation of empty bonus. Binned kd-trees were constructed by the binned SAH and object median. SAH kd-trees were constructed by the full SAH using the longest axis.

	Threads	S/D SAHs	EB	Kitchen	Robot	Museum4	Museum8	Conference
gkDtree	12	Y	partial	3.8	14.8	5.6	852.5	N/A
	12	N	partial	4.2	18.9	5.6	879.2	263.0
	1	Y	partial	15.3	101.2	5.6	852.5	N/A
	1	N	partial	20.9	137.5	5.6	879.2	2139.0
binned kd-tree	1	N/A	Y	368.3	358.3	27.8	350.3	3053.9
	1	N/A	N	441.8	431.4	27.8	356.1	3058.3
SAH kd-tree	1	N/A	Y	632.4	460.7	53.0	942.9	3084.2
	1	N/A	N	715.5	518.4	52.9	930.4	3084.8
Speedup	gkDtree vs SAH kd-tree			166.4×	31.1×	9.4×	1.1×	11.7×
	gkDtree vs binned kd-tree			96.9×	24.2×	5.0×	0.4×	11.6×

times for each frame, respectively.

Table 2 depicts the average update performances of kd-trees with full SAH, kd-trees with binned SAH, and gkDtrees. Kd-trees with full SAH are implemented using the longest axis as described in [13]. Also, kd-trees with binned SAH are implemented as illustrated in [3]. First, up to the eighth level, nodes are built by the object-median heuristic to make 256 subtrees. When we build nodes in the subtrees, we use 32 bins if the number of primitives is larger than 32. If the number of primitives is 32 or lower, we use the exact SAH computation rather than the binned SAH approximation.

According to Table 2, gkDtrees provided 1.1-166.4 times faster update performances than kd-trees. Note that these values do not include the initial gkDtree construction time except for the Conference scene. The initial gkDtree construction time of the Kitchen, Robot, Museum4, and Museum8 scenes, without applying different SAHs to static and dynamic groups, were 569, 83, 48, and 1250 ms, respectively. We considered only the acceleration structure update time after the first frame. Due to the larger leaf size of dynamic groups, applying different SAH schemes to static and dynamic groups showed faster update performance than without applying them. In the Museum scene, there is only one dynamic group, so only one thread was performed for the dynamic group in the Museum scene. Therefore, there was no difference between a single thread and 12 threads. In the Conference scene, instead of using update time, we have

measured construction time because this scene is static. Even a single-threaded gkDtree was about 1.1∼46 times faster than a kd-tree. Such improvement was made possible by gkDtrees' partial update as well as the group-based SAH computation.

Comparisons were made between a gkDtree and a kd-tree with the binned SAH method. We measured the values of the binned SAH method in a single thread. If this method is performed in parallel with our six-core CPU with hyperthreading, it will show approximately 50-60ms in the Kitchen, Robot, and Museum8 scenes, 5ms in the Museum4 scene, and 400-500ms in the Conference scene. Therefore, we conclude that a gkDtree exhibits superior update performance in comparison to the parallel binned SAH method over all the scenes except the Museum8 scene. If an object (or triangle mesh) consists of a lot of primitives such as those in the Museum8 scene, we can apply other parallel tree construction algorithms [3, 5], into the local kd-tree.

These results indicate that a gkDtree provides competitive performance to the state-of-the-art, GPU-based kd-tree construction in [8]. In the Kitchen scene, [8] showed 42.0ms on Geforce GTX 280 and our approach showed only 3.8ms on a 3.33GHz six-core CPU. In the Robot scene, [8] showed 37.0ms and ours showed 14.8ms.

Table 3 depicts statistical tree data as described in [11]. Because we used a leaf size of 8 and a limited tree depth, shallower kd-trees have been built than the kd-trees in [11]. Also, when we apply different SAHs to static and dynamic groups, the expected costs of gkDtrees (C(T)) decreased by 12-20 %. Consequently, the expected costs of the gkDtrees were lower than that of the kd-trees with a full SAH using the longest axis in the Kitchen and Museum scenes.

Table 4 depicts the ray traversal performance on 12 threads. According to the results, gkDtrees showed competitive traversal performance when compared to other tested algorithms. Performance-independent statistics in Table 5 provide further explanations. Kd-tree traversal requires only one ray-box intersection test per ray. These tests are performed between rays and the scene AABB, while gkDtree traversal requires additional ray-box intersection tests with group nodes. On the plus side, group node traversals help reduce the numbers of 'normal' node traversals and ray-primitive intersection tests in a local tree. In contrast to the general kd-tree, primitives in a gkDtree are only split in the local group kd-tree. Thus, redundant ray-primitive intersection tests can be minimized. On the negative side, overlapping local trees may degrade the

Table 3: Statistical data for the generated gkDtrees and kd-trees: N_N, N_L, and N_{NE} are the numbers of nodes, leaf nodes, and non-empty leaf nodes, respectively. N_{AT} is the average number of triangles per non-empty leaf. E_T, E_L, and E_I are the expected numbers of inner-node traversals, leaf visits, and ray-primitive intersections, for a random ray, respectively. C(T) is the expected cost according to Equation 3 in [11].

	S/D SAHs	EB	N_N	N_L	N_{NE}	N_{AT}	E_T	E_L	E_I	C(T)
Kitchen										
gkDtree	Y	partial	931K	465K	421K	4.53	17.33	5.22	11.54	364.34
	N	partial	1016K	508K	449K	4.51	21.38	6.08	13.51	442.19
binned	N/A	Y	285K	142K	101K	10.10	37.72	7.44	14.65	714.54
kd-tree	N/A	N	429K	214K	187K	7.44	35.26	6.94	14.11	667.60
SAH	N/A	Y	112K	56K	43K	11.25	21.37	5.03	13.68	421.15
kd-tree	N/A	N	366K	183K	168K	7.02	23.71	5.31	13.51	461.87
Robot										
gkDtree	Y	partial	199K	99K	91K	7.63	28.35	6.76	15.44	560.39
	N	partial	204K	103K	86K	6.59	35.43	8.51	18.00	701.61
binned	N/A	Y	347K	173K	155K	8.67	47.98	8.54	19.12	890.78
kd-tree	N/A	N	490K	245K	234K	8.10	40.31	7.30	19.12	750.59
SAH	N/A	Y	163K	81K	71K	8.76	28.72	6.34	22.33	557.70
kd-tree	N/A	N	307K	153K	146K	7.93	29.01	6.40	18.01	563.12
Museum4										
gkDtree	Y	partial	13K	6K	4K	9.31	17.42	6.75	24.93	396.35
	N	partial	25K	12K	8K	6.78	20.24	7.49	24.35	453.42
binned	N/A	Y	15K	7K	5K	8.61	26.98	6.08	25.47	526.28
kd-tree	N/A	N	30K	15K	14K	6.20	23.75	5.46	24.45	465.48
SAH	N/A	Y	26K	13K	9K	6.85	22.20	5.53	22.49	438.69
kd-tree	N/A	N	28K	14K	13K	6.14	21.27	5.02	24.58	419.41
Museum8										
gkDtree	Y	partial	173K	86K	83K	49.57	17.80	6.83	30.69	403.54
	N	partial	181K	90K	83K	36.09	20.54	7.54	29.99	458.91
binned	N/A	Y	274K	137K	130K	60.87	34.87	7.02	32.59	663.65
kd-tree	N/A	N	202K	101K	99K	63.48	31.68	6.45	32.31	604.32
SAH	N/A	Y	172K	86K	80K	33.69	22.96	5.30	28.69	450.43
kd-tree	N/A	N	171K	85K	84K	32.40	22.04	5.15	29.62	433.64
Conference										
gkDtree	N/A	partial	1219K	609K	563K	11.95	54.23	14.76	65.76	1108.65
binned	N/A	Y	1590K	795K	728K	17.09	54.44	10.48	105.21	1026.19
kd-tree	N/A	N	1921K	960K	920K	12.60	50.52	9.83	104.19	954.39
SAH	N/A	Y	851K	425K	390K	13.68	43.38	9.48	111.74	840.26
kd-tree	N/A	N	1518K	759K	740K	11.50	39.33	8.58	112.61	761.56

Table 4: Average ray traversal performance including Phong shading and texture mapping (in frames per second) for the test scenes. Note that Phong shading and texture mapping were performed without SIMD instructions.

	S/D SAHs	EB	Kitchen	Robot	Museum4	Museum8	Conference
Single ray traversal							
gkDtree	Y	partial	6.12	5.53	6.03	4.81	N/A
	N	partial	5.67	5.17	4.37	3.79	4.10
binned	N/A	Y	4.74	4.29	5.22	3.48	4.33
kd-tree	N/A	N	5.16	4.45	5.46	3.67	4.30
SAH	N/A	Y	4.93	5.53	5.58	4.32	4.61
kd-tree	N/A	N	5.77	4.96	5.44	4.48	4.62
2x2 packet traversal							
gkDtree	Y	partial	12.47	11.91	14.24	10.58	N/A
	N	partial	11.78	10.76	12.15	9.48	11.30
binned	N/A	Y	9.79	9.13	12.11	8.55	10.44
kd-tree	N/A	N	10.29	9.37	13.15	9.00	10.39
SAH	N/A	Y	10.28	10.43	12.48	10.00	11.22
kd-tree	N/A	N	11.27	10.05	13.48	10.36	11.35

traversal performance.

In the case of single ray traversal, due to overlapping, normal gkDtrees showed 2-22% slower traversal performance than the best case of kd-trees with SAH. However, when we apply different SAHs to static and dynamic groups in the case of dynamic scenes, not only tree update performance but also tree traversal performance were increased. As a result, gkDtrees showed up to 6% faster traversal performance than kd-trees with a full SAH using the longest axis.

In the case of packet traversal, the normal gkDtrees showed slightly faster performance than kd-trees in the Kitchen and Robot scenes. Also, when we apply different SAHs to static and dynamic groups, gkDtrees showed 2-14% faster traversal performance than kd-trees in the all BART scenes. These results indicated that gkDtrees have relatively better performance on packet tracing. We explain the reason as follows. Kd-trees' major advantage is early termination using strict front-to-back traversal [17]. In contrast, gkDtrees only guarantee early termination in a local tree due to overlapping groups. It is the main weakness of gkDtrees. However, packet tracing would reduce the advantage of the early termination because the traversal of a ray packet is terminated only if all rays in the packet finished their traversals [34].

In contrast to a gkDtree, the parallel binned SAH method always decreases

Table 5: Performance-independent statistics with single ray tracing.

	S/D SAHs	EB	Kitchen	Robot	Museum4	Museum8	Conference
Ray-box intersection tests per ray							
gkDtree	Y	partial	9.05	8.48	4.23	4.22	N/A
	N	partial	9.02	8.48	4.24	4.25	11.97
binned	N/A	Y	1.00	1.00	1.00	1.00	1.00
kd-tree	N/A	N	1.00	1.00	1.00	1.00	1.00
SAH	N/A	Y	1.00	1.00	1.00	1.00	1.00
kd-tree	N/A	N	1.00	1.00	1.00	1.00	1.00
Node traversals per ray							
gkDtree	Y	partial	45.19	49.03	43.97	46.78	N/A
	N	partial	50.08	55.75	61.88	64.60	56.53
binned	N/A	Y	62.27	65.99	48.49	68.10	58.59
kd-tree	N/A	N	57.41	59.08	42.09	60.45	57.42
SAH	N/A	Y	48.98	46.67	39.27	42.44	37.73
kd-tree	N/A	N	43.40	47.08	35.73	40.74	35.35
Ray-primitive intersection tests per ray							
gkDtree	Y	partial	9.17	12.76	14.26	17.94	N/A
	N	partial	10.76	13.97	16.54	20.46	27.34
binned	N/A	Y	17.87	22.44	26.37	38.49	30.83
kd-tree	N/A	N	16.47	22.69	23.31	38.11	31.50
SAH	N/A	Y	15.86	16.67	24.92	27.74	34.74
kd-tree	N/A	N	13.92	19.26	23.85	28.26	35.83

the traversal performance as contrasted with kd-trees. This method increases the numbers of both node traversal steps and ray-primitive intersection tests in comparison to the full SAH. Another publication [5] has also reported similar performance degradation with this method. This performance degradation is mainly caused by the object median for the parallel tree build. According to these results, we can expect that our scene graph-based gkDtree would provide higher quality trees than object median-based parallelization in the case of dynamic scenes.

Figure 5 shows the scalability of gkDtree updates with up to 12 threads. The update frame rates on different numbers of threads were normalized by the single threaded frame rate. As mentioned earlier, 7∼12 threads were performed in parallel using hyper-threading because we used a six-core CPU. Hence, the scalability in more than six threads is slowly increased. The scalability depends heavily upon the number of dynamic primitives in the test scenes. For the Robot and Conference scenes, the gkDtrees achieved a nearly linear speed increase up to six threads. This result can be explained by the fact that, as long as a scene

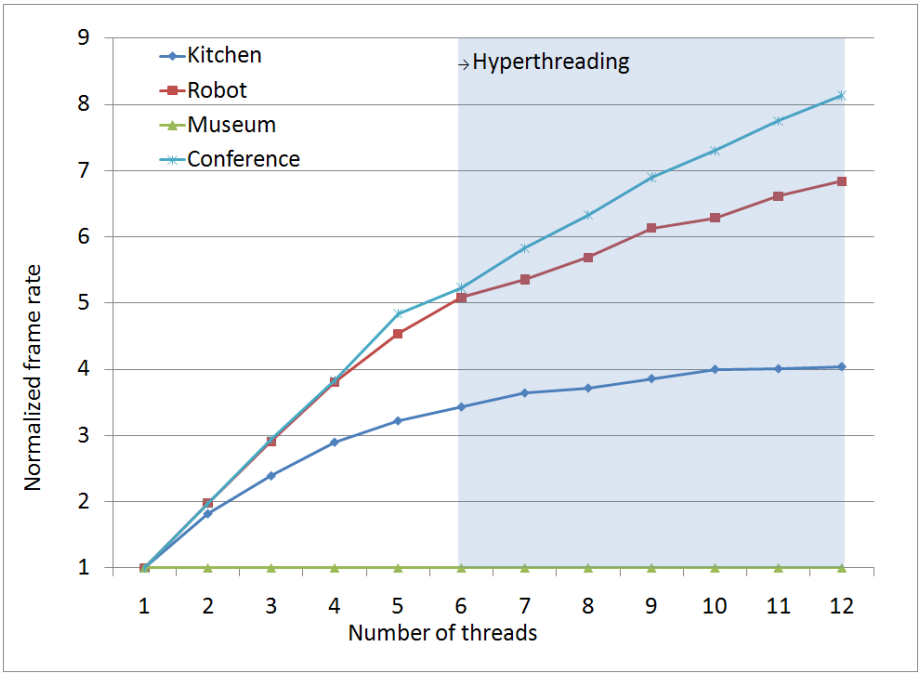

Figure 5: Parallel scalability of gkDtrees up to 12 threads.

has a sufficient number of dynamic groups to feed the simultaneously running threads, all of the threads will tend to remain busy the majority of the time. Hence, we may conclude that the gkDtree is more scalable in complex scenes having a large number of dynamic objects. In the Museum scene, the scalability remains approximately 1.0, although the number of threads increases, because there is only a single dynamic group (an exploding sphere). In this case, a gkDtree can become a hybrid with the geometry decomposition method [3, 5] to attain better scalability.

5. Conclusion and Future Work

This paper proposed a new group-based acceleration data structure called gkDtree for interactive ray tracing of dynamic scenes. The main idea of the gkDtree is to incorporate both the multi-level hierarchy and parallelization approaches to a kd-tree based acceleration structure. For a gkDtree, a scene is partitioned hierarchically using its scene graph. Only the local acceleration structure of each group that contains moving primitives is rebuilt. In order to

rebuild the local acceleration structures in parallel, simple, dependency-free and load-balancing schemes have been introduced.

The experimental results indicated that a gkDtree has up to 166 times faster update performances than a kd-tree for the BART test scenes in a six-core hardware system environment. Particularly for the scenes containing a relatively smaller portion of dynamic primitives (e.g., the BART Kitchen scene), a gkDtree showed very high performance improvement in updates. In addition, a gkDtree even improved the update performance for those scenes with a large number of dynamic groups (e.g., the BART Robot scene). Because many real, dynamic environments, such as game engines, have both static and dynamic objects (e.g., BART scenes), it is expected that our group-based parallel and partial kd-tree construction can be used more widely.

In our future work for this study, we plan to apply the gkDtree to a GPU ray tracer. The amount of data transfer between CPU and GPU should be carefully monitored to improve the overall rendering performance. In addition, we plan to combine the gkDtree with other acceleration techniques, such as the use of precomputed triangle clusters [16]. We believe that this combination would be effective in larger data sets.

6. Acknowledgement

We thank very much Jeong-Soo Park and anonymous reviewers for the valuable comments that help improve our manuscript. The Conference scene is courtesy of Greg Ward.

References

[1] I. Wald, W. R. Mark, J. Gunther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, P. Shirley, State of the art in ray tracing animated scenes, Computer Graphics Forum 28 (6) (2009) 1691–1722.

[2] J. Lext., T. Akenine-Möller, Towards Rapid Reconstruction for Animated Ray Tracing, in: Eurographics 2001–Short Presentations, 2001, pp. 311–318.

[3] M. Shevtsov, A. Soupikov, A. Kapustin, Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes, Computer Graphics Forum 26 (3) (2007) 395–404.

[4] I. Wald, On fast construction of sah-based bounding volume hierarchies, in: Proceedings of IEEE Symposium on Interactive Ray Tracing 2007, 2007, pp. 33–40.

[5] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, J. C. Hart, Parallel SAH k-D tree construction, in: Proceedings of the Confernce on High Performance Graphics 2010, 2010.

[6] K. Zhou, Q. Hou, R. Wang, B. Guo, Real-time kd-tree construction on graphics hardware, ACM Trans. Graph. 27 (5) (2008) 1–11.

[7] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Fast BVH construction on GPUs., Comput. Graph. Forum 28 (2) (2009) 375–384.

[8] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, D. Manocha, Memory-scalable GPU spatial hierarchy construction, IEEE Transactions on Visualization and Computer Graphics 17 (3) (2011) 466–474.

[9] J. Pantaleoni, D. Luebke, HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry, in: Proceedings of the Confernce on High Performance Graphics 2010, 2010.

[10] W. Hunt, W. Mark, D. Fussell, Fast and lazy build of acceleration structures from scene hierarchies, in: Proceedings of IEEE Symposium on Interactive Ray Tracing 2007, 2007, pp. 47–54.

[11] I. Wald, V. Havran, On building fast kd-trees for ray tracing, and on doing that in o(n log n), Proceedings of IEEE Symposium on Interactive Ray Tracing 2006 (2006) 61–69.

[12] D. J. MacDonald, K. S. Booth, Heuristics for ray tracing using space subdivision, The Visual Computer 6 (3) (1990) 153–166.

[13] M. Pharr, G. Humphreys, Physically Based Rendering, 2nd Edition, Elsevier, 2010.

[14] J. Lext, U. Assarsson, T. Moller, BART : A benchmark for animated ray tracing, IEEE compter graphics and applications 21 (2) (2001) 22–31.

[15] W. Hunt, W. Mark, G. Stoll, Fast kd-tree construction with an adaptive error-bounded heuristic, in: Proceedings of IEEE Symposium on Interactive Ray Tracing 2006, 2006, pp. 81–88.

[16] K. Garanzha, The use of precomputed triangle clusters for accelerated ray tracing in dynamic scenes, Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2009) 28 (4) (2009) 1199–1206.

[17] C. Lauterbach, S.-E. Yoon, D. Tuft, D. Manocha, RT-DEFORM: Interactive ray tracing of dynamic scenes using BVH, in: Proceedings of IEEE Symposium on Interactive Ray Tracing 2006, 2006, pp. 39–45.

[18] V. Havran, R. Herzog, H.-P. Seidel, On the fast construction of spatial data structures for ray tracing, in: Proceedings of IEEE Symposium on Interactive Ray Tracing 2006, 2006, pp. 71–80.

[19] I. Wald, S. Boulos, P. Shirley, Ray tracing deformable scenes using dynamic bounding volume hierarchies, ACM Transactions of Graphics 26 (1) (2007) 6.

[20] I. Wald, C. Benthin, P. Slusallek, Distributed interactive ray tracing of dynamic scenes, in: IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003, pp. 77–86.

[21] J. Bikker, Real-time ray tracing through the eyes of a game developer, in: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing, 2007, pp. 1–10.

[22] T. Ize, I. Wald, S. G. Parker, Asynchronous bvh construction for ray tracing dynamic scenes on parallel multi-core architectures, in: In Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization, 2007, pp. 101–108.

[23] S.-E. Yoon, S. Curtis, D. Manocha, Ray tracing dynamic scenes using selective restructuring, in: Proceedings of Eurographics symsposium on rendering 2007, 2007.

[24] T. Ize, I. Wald, C. Robertson, S. Parker, An evaluation of parallel grid construction for ray tracing dynamic scenes, in: Proceedings of IEEE Symposium on Interactive Ray Tracing 2006, 2006, pp. 47–55.

[25] C. Benthin, Realtime ray tracing on current cpu architectures, Ph.d. thesis, Saarland University (Jan. 2006).

[26] S. Popov, J. Gunther, H.-P. Seidel, P. Slusallek, Experiences with streaming construction of sah kd-trees, in: Proceedings of IEEE Symposium on Interactive Ray Tracing 2006, 2006, pp. 89–94.

[27] J. Kalojanov, P. Slusallek, A parallel algorithm for construction of uniform grids, in: Proceedings of the Conference on High Performance Graphics 2009, ACM, 2009, pp. 23–28.

[28] J. Kalojanov, M. Billeter, P. Slusallek, Two-level grids for ray tracing on GPUs, Computer Graphics Forum (EUROGRAPHICS 2011) (to appear).

[29] J. Goldsmith, J. Salmon, Automatic creation of object hierarchies for ray tracing, IEEE Computer Graphics and Applications 7 (5) (1987) 14–20.

[30] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, W. R. Mark, Razor: An architecture for dynamic multiresolution ray tracing, Tech. Rep. 07-52, University of Texas at Austin, Department of Computer Sciences (Jan. 2007).

[31] V. Havran, Heuristic ray shooting algorithms, Ph.d. thesis, Czech Technical University in Prague (Nov. 2000).

[32] J. Amanatides, A. Woo, A fast voxel traversal algorithm for ray tracing, in: EUROGRAPHICS '87, 1987, pp. 3–10.

[33] M. Shevtsov, A. Soupikov, A. Kapustin, Ray-triangle intersection algorithm for modern CPU architecture, in: Proceedings of GraphiCon 2007, 2007.

[34] I. Wald, P. Slusallek, C. Benthin, Interactive distributed ray tracing of highly complex models, in: Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering), 2001, pp. 277–288.

[35] T. Möller, B. Trumbore, Fast, minimum storage ray-triangle intersection, Journal of graphics, GPU, and game tools 2 (1) (1997) 21–28.