Improving Memory Space Efficiency of Kd-tree for Real-time Ray Tracing

B. Choi, B. Chang, and I. Ihm
Department of Computer Science and Engineering, Sogang University, Korea

Abstract

Compared with its competitors such as the bounding volume hierarchy, a drawback of the kd-tree structure is that a large number of triangles are repeatedly duplicated during its construction, which often leads to inefficient, large and tall binary trees with high triangle redundancy. In this paper, we propose a space-efficient kd-tree representation where, unlike commonly used methods, an inner node is allowed to optionally store a reference to a triangle, so highly redundant triangles in a kd-tree can be culled from the leaf nodes and moved to the inner nodes. To avoid the construction of ineffective kd-trees entailing computational inefficiencies due to early, possibly unnecessary, ray-triangle intersection calculations that now have to be performed in the inner nodes during thekd-tree traversal, we present heuristic measures for determining when and how to choose triangles for inner nodes during kd-tree construction. Based on these metrics, we describe how the new form of kd-tree is constructed and stored compactly using a carefully designed data layout. Our experiments with several example scenes showed that our kd-tree representation technique significantly reduced the memory requirements for storing the kd-tree structure, while effectively suppressing the unavoidable frame-rate degradation observed during ray tracing.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

1.1. Background

Thanks to their reliable high performance in the calculation of ray-object intersections, kd-trees are considered one of the most effective acceleration structures for ray tracing. Given a three-dimensional scene, the kd-tree is constructed in a top-down manner by recursively subdividing the scene’s voxel space using axis-aligned splitting planes. The timing performance of the resulting kd-tree is affected greatly by the fundamental decisions of where to position the splitting planes and when to stop the recursive subdivision. The surface-area heuristic (SAH), introduced by MacDonald and Booth [MB90], is generally acknowledged as the best strategy for building high-quality kd-trees [Hav01, Wal04], because its simple cost-prediction model based on the theory of geometric probability [San02] usually facilitates reasonably good decision making.

In the standard kd-tree representation, each inner node stores information about the splitting plane and addresses pointing to its two child nodes, while each leaf node stores a list containing indices referring to the triangles that overlap with the voxel referenced by the leaf. This method is used widely but this representation cannot avoid the following inherent problem that arises during the construction of kd-trees. If a triangle in the voxel penetrates a selected splitting plane when a voxel is being subdivided, this triangle is sorted into both subvoxels redundantly, which increases the actual number of triangles handled by the construction algorithm. If this happens frequently, the leaf nodes in the resulting kd-tree have a high frequency of duplicate references to the same triangles. This is known to be a drawback of the kd-tree structure, especially when compared with the bounding volume hierarchy (BVH), in which triangles are generally not shared by leaf nodes, so the resulting binary trees are usually smaller and shallower than the kd-trees.

Table 1 shows that many duplicate triangles are present in the leaf nodes of kd-trees constructed for five representative scenes using the de facto standard SAH-based construction algorithm [WH06]. The count column shows the number of
triangles in a scene for each interval. Compared with the original number of triangles in each scene, the total numbers of indices stored in the leaf nodes of the respective kd-trees, i.e., the no. of indices, demonstrate the redundancy problem, which is inevitable in the current kd-tree representation and it may lead to inefficient tree structures.

We tested several scenes and found that most of the triangles in the Kitchen scene, i.e., 98,849 out of 101,015 (97.9%), were small and well tessellated. This was a rather good example because there was only a 3.20 times increase in the number of stored references. By contrast, a substantial amount of ill triangles in the Conference scene intersected repeatedly with the splitting planes selected by the SAH metric, which produced a high ratio of 14.41. In this table, the figures in the longest column denote the average length of the longest edges of triangles in the corresponding frequency range. In many cases, these results show that relatively large and/or skinny triangles generally led to excessive redundancy in the tree structure (refer to Figure 1).

The replication of triangles during kd-tree construction means that the acceleration structure often imposes a non-trivial spatial overhead. This is shown in Table 2, where the size of the kd-trees built using a construction algorithm [WH06] and represented with a compact data structure proposed by Wald [Wal04] are compared with the normalized sizes of the geometry data. This analysis and that in the previous table suggest that it would undoubtedly be worth-while to develop a new kd-tree scheme that facilitates a more compact representation by reducing triangle redundancy.

1.2. Our contribution

In this paper, we propose a space-efficient kd-tree representation method that unlike the standard method, permits the optional storage of a reference to a triangle in the inner nodes. The inherent problem of kd-trees that the tree structure easily grows in size due to duplicate references to the geometry is eased markedly by selecting triangles that cause excessive duplication, and storing them in proper inner nodes, which significantly reduces the memory space to store the redundant references in the leaf nodes.

Our new kd-tree representation method requires a slight modification of the standard kd-tree traversal algorithm (e.g., that summarized in [Hav01, Wal04]), where a ray-triangle intersection computation is required each time an inner node containing a triangle reference is visited, and the first hit in a leaf node is compared to that, if any, between the ray and the triangles of the visited inner nodes. This method is sim-
2. Related work

The ray-tracing performance of kd-trees is affected greatly by the way they are constructed, particularly the positioning of the axis-aligned splitting planes during recursive kd-tree construction. Among several possible strategies, the surface area heuristic (SAH) method introduced by MacDonald and Booth [MB90], is generally acknowledged to produce the best kd-trees in terms of ray-tracing speed [Hav01, Wal04]. Several modifications have been made to the SAH-based cost metric [Hav01, HKRS02, Han08, RKJ96, FFD09, CCI12, WGS04, IH11] to further enhance the temporal performance of SAH kd-trees.

From an algorithmic point of view, Wald and Havran introduced an algorithm constructing a kd-tree in $O(n \log n)$ time, which is the theoretical lower bound [WH06]. Hunt et al. [HMS06], Popov et al. [PGSS06], and Shevtsov et al. [SSK07] presented faster, scanning-based algorithms that approximated the SAH cost function with only slight frame-rate degradation. High construction costs were considered to be a key drawback that prevented kd-trees from being used in the rendering of dynamic scenes. However, fast kd-tree construction is now possible via the effective parallelization of the construction process on modern CPUs and GPUs, as reported previously [ZHWG08, CKL*10, WZL11]. Furthermore, considerable attention has been paid to the development of memory-efficient kd-tree layouts and the optimization of the traversal algorithm for specific hardware (for example, refer to a recent review article [HH11]).

As well as improving the temporal performance, the generation of space-efficient kd-tree structures is equally important because it facilitates effective memory management during ray tracing. However, most attempts to reduce the size of acceleration hierarchies and geometries have focused on other structures such as the BVH and its variants. Mahovsky and Wyvill proposed a hierarchical BVH encoding scheme that reduces the memory requirements by storing the six coordinates of the axis-aligned bounding box of a node relative to its parent node using 4-bit or 8-bit unsigned integers [MW06]. Cline et al. also used lower precision numbers and reduced the size of the resultingBVH further by employing a pointer-free heap-like data structure with a high branching factor [CSE06]. Wächter and Keller also generated compact hierarchies using a lazy building technique [WK06].

To render large models, Lauterbach et al. presented a two-level hierarchy model where lower level hierarchies implicitly encoded those generated from carefully constructed triangle strips [LYTM08]. This idea of a two-level structure was also applied by Segovia and Ernst to the lossy encoding of BVH nodes and their geometries on two levels [SE10]. Bauszat et al. further compressed the BVH using a modified hierarchy known as the minimal bounding volume hierarchy [BEM10]. Kim et al. proposed a cluster-based layout-preserving BVH compression technique, which gave effi-
icient random access to the compressed BVHs [KMKY10]. On the other hand, Hou et al. proposed a construction scheme that builds kd-trees/BVHs for large models effectively on the GPU with limited memory [HSZ+11].

Finally, Havran et al. allowed to link an extra tree structure, built for oversized geometries, as a ternary child of an inner node of a hybrid tree blending a spatial kd-tree (SKD-tree) with bounding volume primitives [HHS06]. While their method mainly focused on fast construction of spatial hierarchies, allowing to store ternary tree structures in the inner nodes often degraded the ray-tracing performance significantly. Our method differs in that only a carefully chosen triangle is stored in an inner node of the standard kd-tree, effectively suppressing the unavoidable frame-rate degradation observed during ray tracing. In order to reduce the search space quickly during ray tracing, Zuniga et al. isolated a wide primitive in a voxel as a right child of the corresponding inner node of another hybrid, dual extent tree [ZU06]. However, such strategy usually increased the tree size markedly due to the resulting local unbalance in the tree structure.

3. Construction of space-efficient kd-trees

3.1. Two heuristic metrics for triangle selection

Figure 2 shows the key concept of our method for increasing space efficiency by eliminating highly duplicated references to the same triangles as much as possible. As mentioned in the Introduction, however, storing triangles (i.e., storing references to triangles) in inner nodes may degrade the ray-tracing performance, mainly because it increases the number of unnecessary early calculations of ray-triangle intersections. Thus, the triangles must be selected carefully to greatly reduce the memory requirements while avoiding wasteful computations as much as possible.

Consider a kd-tree \( T \), which is built using a standard SAH-based method, and its subtree \( T_N \), which is rooted in the node \( N \), where the corresponding voxel region is denoted as \( V_N \). The first necessary condition that selects a triangle as a candidate for an inner node \( I \) is the likelihood that for a given random ray the triangle will eventually be tested for a ray-triangle intersection, irrespective of whether it is actually hit by the ray or not, during the traversal of \( T_I \). This condition is obvious because a higher likelihood means there is a lower probability that the early evaluation of the ray-triangle intersection will turn out to be unnecessary.

Let \( T(T_I) \) denote the set of all triangles stored in the leaf nodes of \( T_I \). We also assume that, given a triangle \( t \in T(T_I) \), \( \mathcal{L}(t, T_I) \) represents the set of all leaf nodes of \( T_I \) passed by \( t \), where \( \mathcal{L}(\cdot, T_I) \) specifically denotes the set of all leaf nodes of \( T_I \). We also define \( SA(V) \) as the surface area of the voxel \( V \), so the first surface area-based occupancy measure of the inner node \( I \) is defined as follows:

\[
f_{\text{occ}}(t, T_I) = \frac{\sum_{L \in \mathcal{L}(t, T_I)} \text{SA}(V_L)}{\sum_{L \in \mathcal{L}(\cdot, T_I)} \text{SA}(V_L)}
\]

where the numerator, i.e., the sum of the geometric probability that each leaf node intersecting with the triangle \( t \) will be hit by a random ray, is normalized between 0 and 1. Note that, if a leaf intersecting with the triangle \( t \) is reached during ray tracing, a ray-triangle intersection calculation is incurred with respect to \( t \). Therefore, this occupancy measure provides a good estimate of the chance that a ray-triangle intersection computation for \( t \) occurs during the traversal of \( T_I \). Only storing a triangle with a sufficiently high occupancy measure in the inner node may reduce the timing penalty incurred by unnecessary ray-triangle intersection calculations at inner nodes.

We apply a second requisite condition that a triangle is culled from the leaf nodes of a subtree if its effect in terms of data compression is sufficiently high. Placing a triangle on an inner node inevitably leads to inefficiencies during tree traversal. Thus, we require that a triangle is selected on an inner node only if eliminating the triangle from the leaves results in an adequate reduction in the data size. To implement this condition, we define the following simple frequency measure, which determines how frequently the triangle \( t \) appears in the leaf nodes of \( T_I \):

\[
f_{\text{freq}}(t, T_I) = \frac{|\mathcal{L}(t, T_I)|}{|\mathcal{L}(\cdot, T_I)|}.
\]

Based on these two metrics, we define an oracle function \( \text{pick}_\text{triangle}(T_I) \) as follows, which selects a triangle from \( T_I \) if an eligible one is available, whereas it returns a null value if not. First, we enumerate the triangles in \( T_I \) with an occupancy measure \( f_{\text{occ}}(t, T_I) \) greater than a given occupancy threshold \( \tau_{\text{occ}} \) in a nonincreasing order. We check each triangle until we find one with a frequency measure \( f_{\text{freq}}(t, T_I) \) greater than a preset frequency threshold \( \tau_{\text{freq}} \). A suitable triangle is returned only if one is available.

\begin{figure}
\centering
\includegraphics[width=0.8\textwidth]{figure2.png}
\caption{Augmented inner nodes. Given a subtree, we have the option of storing a reference to a triangle with high redundancy in the root of the subtree instead of leaving multiple copies in the leaf nodes, which greatly reduces the space needed to represent the subtree.}
\end{figure}
3.2. The new kd-tree construction algorithm

Now, a new kd-tree can be constructed by a simple recursive process using our rule for choosing a problematic triangle from a subtree. Starting with a kd-tree \( T \) constructed for the triangle set \( T_5 \) of an input scene \( S \) using a standard SAH-based construction algorithm, we examine a given tree \( T_1 \), whose root node \( I \) is not a leaf, to evaluate the oracle function \( \text{pick\_triangle}(T_1) \). If an eligible triangle is not found, the tree building process stops. However, if a triangle \( t \) is selected from \( T(T_1) \) for culling, where \( T(T_1) \) is the set of triangles in the leaves of \( T_1 \), a new SAH-based kd-tree \( T^*_1 \) is built from \( T(T_1) \) excluding \( \{t\} \), replacing \( T_1 \). After \( t \) is stored in the root node of \( T^*_1 \), the same process is applied recursively to the subtree rooted in each child of the root node of \( T^*_1 \).

Algorithm 1 describes an iterative version of this recursive algorithm where direct stack manipulation provides higher runtime performance. When implementing this algorithm, we need to ensure that we do not generate kd-trees containing excessive amounts of triangles placed on the inner nodes. These trees are favorable for memory reduction, but they may tend to slow down ray tracing markedly because they incur a high frequency of, possibly unnecessary, early ray-triangle intersection calculations on the inner nodes. In our method, we allow a triangle \( t \) to be stored in an inner node \( I \) only if the number of triangles including \( t \), evaluated by the function \( \text{count\_triangles}(I) \) in Line 8, which are placed on inner nodes on the path from the node \( I \) to the root node, does not exceed a preset threshold \( \text{max}_{2\text{rn}} \) (the subscript \( 2\text{rn} \) denotes the triangles to the root node). In general, our preliminary experiments showed that setting \( \text{max}_{2\text{rn}} \) to 4 generated effective kd-trees with a good balance between size and speed.

Algorithm 1 Iterative kd-tree construction

\begin{verbatim}
1: Build an SAH-based kd-tree \( T \) for \( T_5 \);
2: push\_root\_node(\( T \));
3: while (stack is not empty) do
4:   \( I = \text{pop}() \);
5:   if (\( \{t\} = \text{pick\_triangle}(T_1) \) != NULL) then
6:     Build an SAH-based kd-tree \( T^*_1 \) for \( T(T_1) \) excluding \( \{t\} \);
7:     Replace \( T_1 \) with \( T^*_1 \), and place \( t \) on \( \{I\} \);
8:   if (\( \text{count\_triangles}(I) = \text{max}_{2\text{rn}} \)) then
9:     continue; \( \{\text{Skip the subtrees of } T^*_1\} \)
10: end if
11: end if
12: if (\( \text{child\_node}(T^*_1) \) is not a leaf node) then
13:   push\_child\_node(\( T^*_1 \));
14: end if
15: if (\( \text{child\_node}(T^*_1) \) is not a leaf node) then
16:   push\_child\_node(\( T^*_1 \));
17: end if
18: end while
19: return \( T \);
\end{verbatim}

3.3. Kd-tree node layouts for effective storage

To maximize the benefits of our proposed kd-tree scheme, it is important to design effective data layouts for specific types of kd-tree nodes. Our data structure extends the idea of a compact kd-tree representation with efficient caching and prefetching, which was presented in Wald [Wal04], where each inner or leaf node is stored in a unified data layout with 8 bytes. However, our compact kd-tree representation scheme requires that different types of information are additionally stored in the inner and leaf nodes, so the actual data structure is slightly more complicated than that used in [Wal04].

In our framework, the three least significant bits, known as the layout type indicator, are reserved for all data layouts and this determines the type of a given 8-byte node (see Figure 3). If the indicator \( A \) has a value of 000, 001, or 010, the layout represents the standard inner node with no reference to a triangle (this is known as a type I inner node). In this case, similar to [Wal04], the two lower bits of \( A \) and the most significant 32 bits of the layout. \( B \), denote the dimension and the position of the splitting plane, respectively, while the remaining 29-bit field. \( C \), stores the offset of the left child (note that the offset is always a multiple of eight, while the right child always follows the left child).

If the layout type indicator has a flag of 100, 101, or 110, the layout represents the inner node where a triangle reference is stored (type II inner node). The meaning of the other bit fields is the same as the standard inner node, except the 8 bytes indicated by the offset field \( C \) store information for the reference, which are followed by the left and right children. The reference to a triangle is specified in the T-reference node with a layout type indicator value of 111, where an unsigned integer type index in the upper 32 bits of the layout, \( D \), points to a triangle in the array of triangles (see Figure 3(b)). The lower 32 bits including the indicator are not used currently, but they are allocated for an 8-byte alignment (note that the 3-bit field of the indicator is reserved for a consistent data structure, but it is not essential). They are not used at present, but these bits could be explored in a future extension. For example, if the storage of two triangles was permitted per inner node, the index to the extra triangle could be stored there.

Similar to inner nodes, our scheme supports two different types of leaf nodes, which leads to a significant additional reduction in the memory space required to store the indices of the triangles found in the leaves. The leaf nodes share the same layout type indicator, i.e., 011, but they are categorized based on the 2-bit leaf-node indicator. \( E \). If a leaf node has the indicator value 00 (refer to Figure 3(c)), it is found in the 4-byte mode, while information on the list of triangles for a leaf node is found in a 4-byte triangle index list: i.e., an array of 4-byte unsigned integers specifying the indices in the array of triangles. This format is basically the same as the data layout in [Wal04] where the 32 bits, \( F \), and
Figure 3: Data layouts for kd-tree nodes. As in [Wal04], a unified layout based on 8 bytes is used for efficient memory access, which represents various types of kd-tree nodes. In our scheme, the type of node is categorized based on the 3-bit layout indicator $A$ and the 2-bit leaf-mode indicator $E$.

(a) An inner node:
- $A = 000/001/010$ (type I), or
- $A = 100/101/110$ (type II)

(b) A T-reference node:
- $A = 111$

(c) A leaf node (4-byte mode):
- $A = 011$ and $E = 00$

(d) A leaf node (2-byte mode):
- $A = 011$ and $E = 01$

Figure 4: Example scenes and the camera views tested.
fect the construction of the new kd-trees: i.e., two thresholds, \( \tau_{\text{occu}} \) and \( \tau_{\text{freq}} \), control the selection of a triangle from a given subtree, while an integer, \( \text{max}_{2rn} \), limits the maximum number of inner nodes along a path to the root node that can hold a triangle. The graphs shown in Figure 5 illustrate typical patterns of the performance variation found with our kd-trees with different combinations of \( \tau_{\text{occu}} \) and \( \tau_{\text{freq}} \). First, for a given occupancy threshold (occupancy in the graph), the effect of kd-tree-size reduction declined as the frequency threshold increased (frequency in the graph). This was obvious because a higher frequency threshold would make it difficult to pick triangles for inner nodes. Second, the same trend was observed when increasing the occupancy threshold, although this was less obvious, which suggests that the reduction of the kd-tree size relative to a standard kd-tree was more sensitive to the frequency parameter than the occupancy parameter. Third, the frame-rate degradation when an image was ray-traced using a new kd-tree also decreased as the two threshold values increased. This phenomenon is also easy to understand because the sparse distribution of triangles on the inner nodes due to large threshold values minimizes the waste of computational resources caused by unnecessary early ray-triangle intersection calculations. Therefore, it is particularly important to set the occupancy threshold at a sufficiently high value to maintain the frame-rate degradation at a consistently low level.

Given these results, we tested two different combinations of the two thresholds where \( \text{max}_{2rn} \) was set to 4. In the first, a higher priority was given to a reduction of the frame-rate degradation: i.e., \( (\tau_{\text{occu}}, \tau_{\text{freq}}) = (0.9, 0.7) \). In the second, we aimed to achieve greater kd-tree-size reduction while maintaining the frame-rate degradation at an acceptably low level: i.e., \( (0.5, 0.4) \). The two subtables in Table 3 show the experimental results, where ours \((f)\) and ours \((s)\) correspond to the kd-trees for the first and second combinations, respectively. Table 3(a) shows that the number of kd-tree nodes decreased markedly because highly redundant triangles were moved from leaf nodes to inner nodes so the kd-trees were shorter in height, which is discussed later, and they also required less memory space for storage (the number of triangles stored in the inner nodes are specified in the \( t\text{-inner} \) column). In addition, the use of less restrictive thresholds allowed us to control the construction process effectively to create a kd-tree with a smaller number of nodes.

As expected, culling redundant triangles from the leaf nodes significantly increased spatial coherence in the triangle list between the triangles intersecting with a common leaf node, which facilitated the storage of a substantial proportion of the triangle indices in the 2-byte mode (compare the figures in the 4-byte mode and 2-byte mode columns). The 2-byte mode technique could also be used for storing a standard kd-tree, but its effect was less significant due to worse locality of triangle reference (observe the decrease in the number of triangle indices in the total column). Overall, we achieved a kd-tree-size reduction of 26.4% to 46.1% during the fast rendering-preferred kd-trees (ours \((f)\)) and 32.2% to 56.4% for the small size-preferred kd-trees (ours \((s)\)).

By contrast, the second table \((b)\) of Table 3 shows the temporal performance of the new kd-trees in terms of frame-rate variation when a 1024 \( \times \) 1024 image was rendered by full ray tracing. The timing results showed that the application of the fast rendering-preferred parameters (ours \((f)\)) resulted in less than 3.8% (the Soda Hall scene) to 6.9% (the Power Plant scene) frame-rate degradation compared with the standard kd-trees, which was quite encouraging given that the storage of triangles in the inner nodes may incur unnecessary ray-triangle intersection computations and it also degrades the parallel performance of kd-tree traversal on the CPU and GPU because of the introduction of additional branch tests in the inner nodes. When size reduction was preferred (ours \((s)\)), the frame-rate drop increased due mainly to the increased number of early ray-triangle intersection calculations, although it was usually below 9.8% (the Conference scene) to 12.5% (the San Miguel scene), which may be acceptable in many situations given the reduced memory sizes.

Note that culling triangles from the leaf nodes also contracted the kd-trees. The two example distributions shown in Figure 6 illustrate the common patterns of changes in the frequencies of leaf node depths when a different type of kd-tree was employed. First, the maximum depth, i.e., the height of the kd-tree, decreased from 181 (the standard kd-tree) to 87 (the fast rendering-preferred kd-tree) and 71 (the small size-preferred kd-tree) for the Soda Hall scene, and from 75 to 62 and 57, respectively, for the San Miguel scene. Second, the peak of the distribution tended to move slightly to the left, where significant frequency reductions were made. Naturally, the shorter structures decreased the kd-tree traversal cost during ray tracing. Frequently, we observed that the frame-rate drops with the new kd-trees were markedly smaller on the GPU than the CPU, particularly in the Conference and Soda Hall scenes (see Table 3 again). This may be because the kd-tree contraction had a more positive impact on the GPU ray-tracer, probably because the GPU is generally more vulnerable to frequent, costly divergent branches during kd-tree traversal. However, this phenomenon appeared to be quite complex.

Finally, we achieved very good performance when the value of the \( \text{max}_{2rn} \) parameter was set to 4, although it was not always the best, but it provided a balance between the ray-tracing speed and kd-tree-size reduction. If a larger value was used, the additional compression effect was generally insignificant, which was probably because the fixed \( \tau_{\text{occu}} \) and \( \tau_{\text{freq}} \) values tended to restrict the selection of additional triangles from subtrees at lower levels. The kd-trees usually behaved rather unexpectedly if a value less than 4 was used.

5. Concluding remarks

The main aim of this work was to select triangles that were duplicated in an excessive number of the leaf node voxels.
Figure 5: The effects of varying the occupancy and frequency thresholds on the spatial and temporal performances of the new kd-trees. The experimental results obtained for the Soda Hall scene where \( \max t_{2m} \) was set to 4 show the typical variations in the performance patterns. The kd-tree size reduction represents the memory size reduction with the new kd-tree relative to the standard form, while the frame-rate degradation indicates the corresponding slowdown rate for the ray-tracing frame rate.

Figure 6: Contraction of kd-trees (\( \max t_{2m} = 4 \)). We compared the frequency distribution of the leaf node depths of the three types of kd-trees and there was a clear reduction in height when the new kd-tree representation was applied.

Table 4: Kd-tree construction times (sec.). The construction algorithms were run on a 3.1 GHz Intel Xeon eight-core CPU, but the runtimes could be reduced greatly by using GPU algorithms such as \([ZHWG08, HSZ'11, WZL11]\).

<table>
<thead>
<tr>
<th>method</th>
<th>Kitc.</th>
<th>Conf.</th>
<th>Soda H.</th>
<th>San M.</th>
<th>Pow. P.</th>
</tr>
</thead>
<tbody>
<tr>
<td>std.</td>
<td>1.4</td>
<td>2.7</td>
<td>27.4</td>
<td>249.2</td>
<td>286.0</td>
</tr>
<tr>
<td>ours (f)</td>
<td>5.2</td>
<td>8.6</td>
<td>54.4</td>
<td>1,671.5</td>
<td>2,283.6</td>
</tr>
<tr>
<td>ours (s)</td>
<td>5.0</td>
<td>7.0</td>
<td>60.2</td>
<td>1,323.8</td>
<td>1,825.6</td>
</tr>
</tbody>
</table>

indicated that our method provided a significant kd-tree-size reduction while avoiding any serious degradation of the timing performance by carefully selecting the triangles.

As implied in Table 4, our method is currently limited to static scenes because the kd-tree construction algorithm requires the repeated application of the standard SAH-based construction process \([WH06]\), which is needed for estimating the occupancy of triangles in a voxel. The occupancy measure might also be approximated in terms of the area of the triangle clipped to the current voxel (the theory of geometric probability \([San02]\) states that the probability of a triangle being hit by a random ray is proportional to the area of the triangle). If we could develop an effective occupancy measure that does not require the construction of an SAH-based kd-tree, a simple modification of the standard kd-tree construction algorithm might generate space-efficient kd-trees very quickly.

Dynamic scenes are used widely nowadays, but the construction of space-efficient kd-trees is still important for static scenes, particularly when large ones are ray-traced. We tested a larger scene that was built by duplicating the
Power Plant scene three times (38.2 million triangles), and, similar to the results in the previous section, achieved a kd-tree-size reduction of 35.2% and 36.9% (from 2,181.9 MB to 1,414.1 MB and 1,377.2 MB) for two different threshold pairs, while the respective frame-rate degradation was 2.9% and 5.8% on average (the single-thread mode) for three different views, which indicates the potential of our method for very complex scenes. Note that our method only encodes the acceleration structure but the scene geometry also requires a substantial amount of memory space. Therefore, combining our technique with proper quantization and/or geometry compression techniques would have a great synergistic effect for the ray tracing of very large polygonal models.

Acknowledgments
This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MOE) (No. 2012R1A1A2008958). The test scenes are courtesy of Joachim Helenklaken (Kitchen), Anat Grynberg and Greg Ward (Conference), the UC Berkeley Walk-through Group (Soda Hall), Guillermo M. Leal Llaguno (San Miguel), and the UNC GAMMA group (Power Plant).

References

© 2013 The Author(s)
Computer Graphics Forum © 2013 The Eurographics Association and John Wiley & Sons Ltd
### Table 3: Spatial and temporal performance of the new kd-trees.

Scenes with low to high geometric complexity were used to compare the new kd-tree construction with the standard SAH-based kd-tree.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Tree representation</th>
<th>Camera view 1</th>
<th>Camera view 2</th>
<th>Camera view 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPU1</td>
<td>CPU12</td>
<td>GPU</td>
</tr>
<tr>
<td>Kitchen (101,015)</td>
<td>std.</td>
<td>0.867</td>
<td>8.621</td>
<td>16.448</td>
</tr>
<tr>
<td></td>
<td>ours (f)</td>
<td>0.854</td>
<td>4.030</td>
<td>15.971</td>
</tr>
<tr>
<td></td>
<td>ours (s)</td>
<td>0.794</td>
<td>7.874</td>
<td>15.099</td>
</tr>
<tr>
<td></td>
<td>ours (f)</td>
<td>1.828</td>
<td>3.812</td>
<td>10.901</td>
</tr>
<tr>
<td></td>
<td>ours (s)</td>
<td>1.727</td>
<td>17.241</td>
<td>23.085</td>
</tr>
<tr>
<td></td>
<td>ours (f)</td>
<td>1.209</td>
<td>10.998</td>
<td>14.397</td>
</tr>
<tr>
<td></td>
<td>ours (s)</td>
<td>1.135</td>
<td>10.309</td>
<td>13.687</td>
</tr>
<tr>
<td>San Miguel (10,500,551)</td>
<td>std.</td>
<td>0.102</td>
<td>1.016</td>
<td>2.576*</td>
</tr>
<tr>
<td></td>
<td>ours (f)</td>
<td>0.097</td>
<td>0.965</td>
<td>2.473*</td>
</tr>
<tr>
<td></td>
<td>ours (s)</td>
<td>0.091</td>
<td>0.907</td>
<td>2.307*</td>
</tr>
<tr>
<td>Power Plant (12,748,510)</td>
<td>std.</td>
<td>0.421</td>
<td>3.497</td>
<td>11.458*</td>
</tr>
<tr>
<td></td>
<td>ours (f)</td>
<td>0.424</td>
<td>3.401</td>
<td>11.173*</td>
</tr>
<tr>
<td></td>
<td>ours (s)</td>
<td>0.399</td>
<td>3.185</td>
<td>10.460*</td>
</tr>
</tbody>
</table>

(a) Statistics related to the kd-tree size. In this table, t-inner indicates the new type of inner nodes that hold references to triangles, while the figures in parentheses in the rightmost column represent the size reduction of the new kd-tree relative to the standard SAH-based kd-tree.

(b) Statistics related to the ray-tracing time (fps). For the CPU, the timings were measured in a single-thread mode (CPU1) and a 12-thread mode (CPU12) using dual 6-core CPUs. The figures in parentheses represent the increase in the ray-tracing time with the new kd-tree relative to the standard SAH-based kd-tree. The frame rates marked with an asterisk were measured using an NVIDIA Quadro 6000 GPU because the standard SAH trees and the geometry data would not fit into the memory of the NVIDIA GeForce GTX 580 GPU.