Ray-Box Culling for Tree Structures

JAE-HO NAH[1], WOO-CHAN PARK[2], YOON-SIG KANG[1], AND TACK-DON HAN[1]
[1]*Department of Computer Science*
Yonsei University
Seoul, 120-749 Korea
[2]*Department of Computer Engineering*
Sejong University
Seoul, 43-747 Korea

Ray-primitive intersection tests are the most important operations in ray tracing. The ray-box culling algorithm was presented to accelerate these intersection tests on grid structures, but this algorithm has not been widely used due to the poor ray traversal performance of the grid structures themselves. In this paper, we demonstrate how to apply this algorithm to tree structures and investigate its efficiency in terms of tree construction time and ray traversal performance. Experimental results show that our approach achieves up to 1.15× faster ray tracing performance and up to 1.22× faster total rendering performance, including tree construction and ray tracing.

Keywords: ray tracing, ray-box culling, ray-primitive intersection tests, kd-trees, primitive culling

1. INTRODUCTION

Ray tracing [1] naturally supports various global illumination effects, such as reflection, refraction, and shadows. Thus, it has been widely used in high-quality offline rendering applications. However, the high computation cost of ray tracing is a major obstacle to its use in real-time rendering. Currently, many researchers are studying ray tracing acceleration techniques to overcome this obstacle.

Most ray tracers use acceleration structures, such as grids, kd-trees, and bounding volume hierarchies (BVHs) for fast ray tracing. Uniform grids subdivide space into equal-sized cells. Kd-trees adaptively partition space by using axis-aligned splitting planes. BVHs are hierarchical tree structures constructed by object partitioning.

If no acceleration structures are present, each ray should test the intersections with all primitives in the scene to find the nearest hit point. A primitive is a simplest geometric object that the rendering system can handle, such as a triangle [21]. By using acceleration structures, this computation is greatly reduced. When an acceleration structure is used, traversal is performed prior to the ray-primitive intersection tests. Traversal is the process of visiting nodes of tree structures or visiting cells of grid structures. When a ray visits either a leaf node or a grid cell through traversal, intersection tests between a ray and the primitives in either the leaf node or the grid cell are executed to find the hit point. That is, each ray tests the intersections with the smaller subset of primitives as calculated by traversal.

Most recent ray tracing acceleration algorithms have focused on accelerating tra-

Received August 5, 2011; revised January 20, 2012
Communicated by Tack-Don Han.
* This work was supported by the Students' Association of the Graduate School of Yonsei University funded by the Graduate School of Yonsei University.

versal. In particular, coherent grid traversal [18], the multi-level ray tracing algorithm [19], and dynamic bounding volume hierarchies [15] have achieved interactive ray casting on commodity PCs by using ray packets [8].

Although the use of acceleration structures reduces the number of ray-primitive intersection tests, the cost of these required intersection tests is still high. In other words, the performance of ray tracing is heavily influenced by both traversal and the intersection tests [2]. According to Benthin [2], speeding up intersection tests has become increasingly important. Therefore, additional acceleration techniques that decrease the number of intersection tests are useful for faster ray tracing.

A few approaches are capable of accelerating intersection tests. These approaches can be divided into three categories: single-ray-based primitive culling [3, 20], packet-based primitive culling [4, 5, 6, 18], and mailboxing [2, 7, 8, 9]. Both types of primitive culling are substitutes for expensive and unnecessary ray-primitive intersection tests, allowing the overall rendering speed to increase. Mailboxing reduces the duplication of ray-primitive intersection tests by recording the results of previous intersection tests.

Among these approaches, we shed new light on the ray-box culling algorithm [3], which is a typical type of single-ray-based primitive culling. We chose to highlight this method for several reasons. First, packet-based approaches are useful for coherent rays, but high-quality image synthesis by ray tracing requires many incoherent secondary rays. If the rays in a ray packet are incoherent, frustum (or shaft) culling can be ineffective because a large frustum (or shaft) is constructed. Second, there has been no attempt to apply the ray-box culling method to tree structures. Tree structures, such as kd-trees and BVHs, are known to provide greater rendering performance than grid structures [10]. However, use of the ray-box culling algorithm has only been presented for grid structures. Therefore, the ray-box culling approach has not been widely used for ray tracing despite the fact that it was introduced more than 20 years ago.

In this paper, we present a detailed method that can be used to apply the ray-box culling algorithm to tree structures. We also investigate the effect of this method from two perspectives: rendering performance and tree construction time. The tree construction time is very important in the rendering of dynamic scenes because acceleration structures, such as trees, should be updated at each frame.

After conducting experiments with the physically based rendering toolkit (PBRT) [11], two important results were obtained. First, the ray-box culling method increased the rendering performance regardless of the degree of ray coherence. Second, it prevents a significant decrease in the ray tracing performance when shallow trees were used. Because shallow trees require less construction time than deeper trees, our approach can provide a good trade-off between tree construction time and ray tracing time. Also, the memory footprint can be reduced by using shallow trees.

The remainder of this paper is organized as follows. In Section 2, we briefly review studies related to accelerating intersection tests. In Section 3, we present detailed information about how to apply the ray-box culling algorithm to tree structures. In Section 4, we describe the experimental results obtained from using the PBRT system. In Section 5, we conclude the paper.

2. RELATED WORK

In this section, we provide a short overview of intersection acceleration techniques. The common goal of these techniques is to reduce the number of ray-primitive intersection tests during ray traversal.

2.1 Primitive Culling

Culling algorithms that are used to reduce the number of ray-primitive intersection tests can be divided into two categories: single-ray-based culling and packet-based culling. The ray-box culling algorithm [3] is included the first category. It uses overlap tests between a ray box and the primitive's bounding box. A ray box is constructed using a grid cell and the ray's t values on the cell. Because both of these boxes are axis-aligned bounding boxes (AABBs), in Section 3 we refer to the ray box as rayAABB and the primitive's bounding box as primitiveAABB. In an extended study of the ray-box culling algorithm, Woo [20] presented a dynamic ray bounding box to increase the efficiency of the ray-box culling method.

The second approach, packet-based primitive culling, culls entire ray packets against a primitive. SIMD (single instruction, multiple data) shaft culling [4] uses four corner shaft rays. Coherent grid traversal [18] also uses this approach. The interval arithmetic test [5] culls ray packets using ray intervals. Vertex culling [6] tests the intersection between vertices of a triangle and the planes of a packet's frustum. This method creates a transient frustum to increase the culling rates when reaching a leaf.

2.2 Mailboxing

Mailboxing [7] is another optimization technique that can reduce the number of ray-primitive intersection tests. In spatial subdivision structures, such as grids and kd-trees, primitives can overlap multiple leaves. As such, mailboxing adds a mailbox to each object to prevent the duplication of ray-primitive intersection tests. The mailbox serves as a space to store the ID of the last ray that was tested against each primitive. In parallel environments, memory writing to update mailboxes causes problems. To overcome this, improved mailboxing algorithms have been presented. Hashed mailboxing [2, 8] retains a small hash table in the thread-local memory. Inverse mailboxing [9] stores the last visited primitive IDs on a ray packet. The ray-box culling algorithm can be combined with mailboxing.

2.3 Split Clipping

Split clipping [12] was introduced to reduce the number of ray-primitive intersection tests for kd-trees. Because split clipping provides tighter bounding boxes for primitives straddling the split planes, the number of primitive references in the leaves is effectively reduced. Some studies have extended this method to BVHs to decrease the overlap between BVs. In early split clipping [13], bounding boxes of large primitives are refined before BVH construction. Split BVHs [14] are constructed through spatial splits.

3. RAY-BOX CULLING FOR TREE STRUCTURES

3.1 Overview

In this section, we discuss how to extend the ray-box culling algorithm [3] to tree structures. The ray-box culling method is performed during the leaf traversal stage. The traditional leaf traversal process is as follows. Non-shadow rays must find the nearest hit point. In this case, intersection tests are performed between the ray and all primitives in the leaf. When a shadow ray finds the hit primitive, the tracing of the ray is aborted.

The ray-box culling algorithm inserts the following three steps into the leaf traversal process: rayAABB construction, an AABB/AABB overlap test, and the updating of rayAABB. Algorithm 1 describes these steps. First, we determine the minimum number of primitives to enable the ray-box culling method (line 2). For further experiments, this value is set to 2. Next, the rayAABB is constructed at the beginning of the leaf traversal process (line 3). Overlap tests between rayAABB and the primitiveAABBs are then performed before the ray-primitive intersection tests are conducted (lines 5-8). PrimitiveAABBs are constructed during the acceleration structure construction stage. If two of the AABBs are not overlapped, the intersection test between the ray and the primitive is not executed. If the hit primitive is found in the leaf and the ray type is a non-shadow ray, the rayAABB is updated to reduce the possibility of overlap between the rayAABBs and the rest of the primitiveAABBs (line 12). Of course, updating is performed only if there are additional rest primitives that need to be tested.

Algorithm 1: Leaf traversal with the ray-box culling. Grey lines indicate the culling parts.

```
function LeafTraversal(ray,tmin,tmax,node)
01. n = node.numPrims();
02. bCull = (n>PRIM_NUM)? true:false;
03. if (bCull == true) then rayAABB = CreateRayAABB(ray, tmin, tmax);
04. for (i=0; i<n; i++)
05.   if (bCull == true) then
06.     primAABB = LoadPrimAABB(node.primList[i]);
07.     if (AABBTest(rayAABB, primAABB)==false) then continue;
08.   end if
09.   prim = LoadPrim(node.primList[i]);
10.   if (intersectionTest(ray, prim)==true) then
11.     if (ray.type==SHADOW_RAY) then break;
12.     if (bCull == true && n-i > 1 ) then rayAABB = UpdateRayAABB(ray);
13.   end if
14. end for
```

3.2 PrimitiveAABB Construction

Figure 1 illustrates the primitiveAABBs. If the initial primitive's bounding boxes are used in the construction of acceleration structures [11, 15], the primitiveAABBs are identical to the initial primitive's bounding boxes (Figure 1, left). In this case, there are no additional calculation costs. Because the calculated primitiveAABBs should be kept in memory, the ray box method requires 24 bytes per primitive to store the min-max values of the three axes.

Smaller primitiveAABBs help increase culling efficiency. To obtain smaller primitiveAABBs, split clipping can be applied (Figure 1, right). However, one disadvantage of split clipping is that it increases the kd-tree build time by 2.39× [12]. Additionally, the number of primitiveAABBs will increase because the number of primitiveAABBs is identical to the number of primitive references, not the number of primitives. Thus, we did not consider split clipping for subsequent experiments.

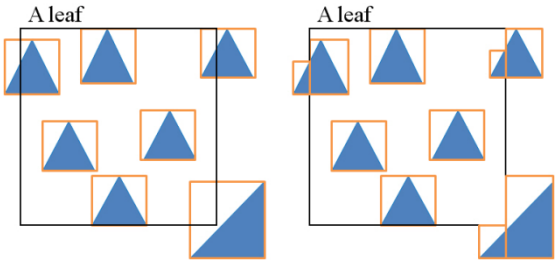


Figure 1. (Left) primitiveAABBs without split clipping; (Right) primitiveAABBs with split clipping.

3.3 RayAABB Construction

Figure 2 illustrates the rayAABB construction process. The rayAABB is an axis-aligned box that bounds the area pierced by a ray. Algorithm 2 describes the rayAABB construction process. A ray with origin o and direction vector d is defined as equation (1):

$$r(t) = o + t \cdot d \qquad (1)$$

When a ray visits the leaves, there are two t values: t^{min} and t^{max}. The first (t^{min}) is the t value at the entry point and the second (t^{max}) is the t value at the exit point. By substituting t^{min} and t^{max} for t in equation (1), we gain the following two intersection points between the ray and the leaf: p^{min} and p^{max}. The first (p^{min}) is the entry point and the second (p^{max}) is the exit point.

If the ray direction sign on an axis is negative, the p^{min} value on the axis is greater than the p^{max} value. Therefore, the ray direction signs should be checked to obtain the precise minimum and maximum AABB values ($rayAABB^{min}$ and $rayAABB^{max}$ in Figure 2) of the three axes. We also add small epsilon values to the calculated rayAABB to prevent visual artifacts that could result due to floating point errors. According to the experimental results, ± 0.0001 is suitable for the epsilon values.

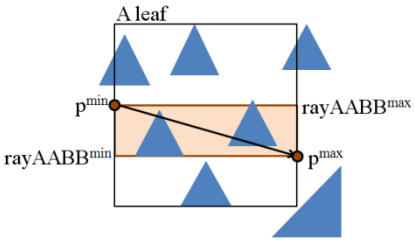


Figure 2. Construction of a rayAABB

Algorithm 2: RayAABB construction

```
function createRayAABB(ray,tmin,tmax)
01. for (axis=0; axis<3; axis++)
02.   pmin[axis] = ray.o[axis] + tmin * ray.d[axis];
03.   pmax[axis] = ray.o[axis] + tmax * ray.d[axis];
04.   if (ray.d[axis]>=0) then
05.     rayAABB.min[axis] = pmin[axis] - EPSILON;
06.     rayAABB.max[axis] = pmax[axis] + EPSILON;
07.   else then
08.     rayAABB.min[axis] = pmax[axis] - EPSILON;
09.     rayAABB.max[axis] = pmin[axis] + EPSILON;
10.   end if
11. end for
12. return rayAABB;
```

3.4 The AABB/AABB Overlap Test

After obtaining the primitiveAABBs and the rayAABB, the culling process is performed using the two AABBs as shown in Figure 3 (left). This culling routine uses a very simple AABB/AABB overlap test, as described in Algorithm 3. If the test result is true, an intersection between the ray and the primitive is possible. If this occurs, a ray-primitive intersection test is performed.

Algorithm 3: AABB/AABB overlap test

```
function AABBTest(rayAABB, primAABB)
01. if (rayAABB.min[0]>primAABB.max[0] ||
      rayAABB.max[0]<primAABB.min[0] ||
      rayAABB.min[1]>primAABB.max[1] ||
      rayAABB.max[1]<primAABB.min[1] ||
      rayAABB.min[2]>primAABB.max[2] ||
      rayAABB.max[2]<primAABB.min[2]) then
02.   return false;
03. end if
04. return true;
```

3.5 Updating of RayAABBs

Smaller rayAABBs help increase the efficiency of culling; in this way they are similar to primitiveAABBs. To exploit this feature, the rayAABB could be updated to a smaller size when a ray hits a primitive (Figure 3, right) as described in [20]. The updating cost is half of the rayAABB construction cost because the updating process only requires the updating of the t^{max} value, as described in Algorithm 4. If we calculate the position of a hit point immediately after the hit point is found, then the updating cost is zero because the position of the hit point is the same as the updated vertex of the rayAABB.

Algorithm 4: updating of a rayAABB

```
function updateAABB(ray)
01. for (axis=0; axis<3; axis++)
02.    phit[axis] = ray.o[axis] + ray.tmax * ray.d[axis];
03.    if (ray.d[axis]>=0) then rayAABB.max[axis] = phit[axis] + EPSILON;
04.    else then rayAABB.min[axis] = phit[axis] - EPSILON;
05.    end if
06. end for
07. return rayAABB;
```

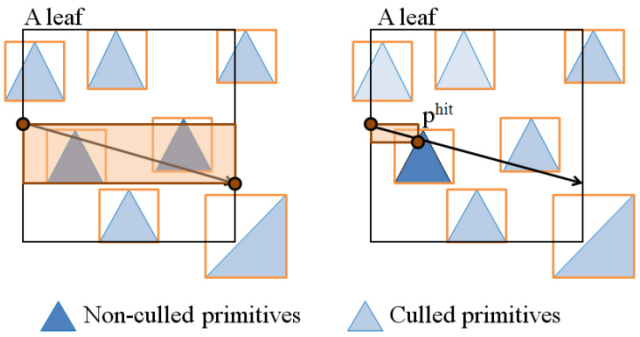


Figure 3. (Left) rayAABB/primitiveAABB overlap test; (Right) updating of a rayAABB

4. EXPERIMENTAL SETUP AND RESULTS

4.1 Experimental Setup

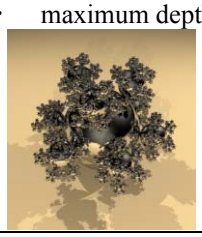
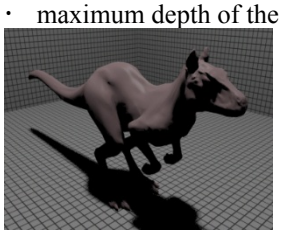
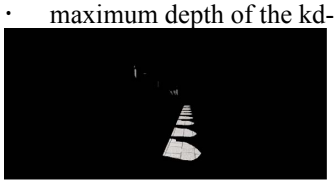
To evaluate our approaches, we implemented the ray-box culling algorithm using PBRT v2 [11]. All tests were performed on a 3.3GHz Intel Core i7 980X with 6GB DDR3 RAM. In order to fully exploit six hyper-threaded cores in a Core i7 980X, we used 12 threads for rendering. Because PBRT does not support parallel tree construction, multi-threading was only used for parallel rendering after single-threaded tree construction.

The experiment was setup in PBRT as follows. For the acceleration structures, kd-trees were used. In the PBRT's surface area heuristic (SAH) [16] construction, the maximum number of primitives in the leaf node (*maxPrims)* was 1, 8, 16, and 32 of primitives. When a tree is constructed in this way, a leaf node is created if the number of

primitives is less than the *maxPrims*. Using this configuration, we tested the efficiency of the ray-box culling algorithm with shallow trees. The number of ray-primitive intersection tests is proportional to the leaf sizes. In contrast, the tree build costs and traversal costs are inversely proportional to the leaf sizes. Therefore, we can expect that our approach will be advantageous in dynamic scenes despite the fact that the PBRT system does not support dynamic scenes. Note that the actual leaf size can be larger than the set leaf size because the max tree depth and the SAH values also make leaf nodes.

We used four benchmark scenes: Balls, Killeroo, Buddha, and Sponza. Table 1 includes details of these scenes. The kd-tree max depth in each scene was determined by the PBRT heuristic: $8 + 1.3 \times \log_2$ (the number of primitives). The Sponza scene was rendered with different settings (direct lighting and path tracing) to measure the effect of our approach on ray coherence.

Table 1: Details of the four PBRT scenes.

<table>
<tr>
<td>Balls
· 2 triangles + 7,381 spheres
· direct lighting (2 samples)
· 3 point light sources
· 900×900 resolution
· maximum depth of the kd-tree: 25
  </td>
<td>Killeroo
· 33,271 triangles
· direct lighting (4 samples)
· 1 area light source (1 sample)
· 684×513 resolution
· maximum depth of the kd-tree: 27
  </td>
</tr>
<tr>
<td>Buddha
· 1,087,721 triangles
· direct lighting (4 samples)
· 1 area light source (8 samples)
· 256×600 resolution
· maximum depth of the kd-tree: 34
  </td>
<td>Sponza
· 66,454 triangles
· direct lighting (1 sample) and
path tracing (8 samples, max depth 3)
· 1 point light source
· 750×350 resolution
· maximum depth of the kd-tree: 29
 
(direct lighting)
 
(path tracing)</td>
</tr>
</table>

4.2 Results and Analysis

We compared the experimental results with and without the ray-box culling algorithm. The results included the acceleration structure build time, the ray tracing time, and various performance-independent statistics. Table 2 describes the results in detail. Figure 4 describes the peak performance comparisons based on the results. A result of 100 percent means that peak performance was achieved without the ray-box culling algorithm. Figure 5 compares relative tree build time and ray tracing time by changing leaf sizes. Figure 6 depicts the culling efficiency of the ray-box culling algorithm. Finally, in Table 3 and Figure 7, we analyze the memory footprint.

Table 2. Experimental results using the PBRT system. Notation: T_B – time required to build the kd-tree; N_T – the number of traversals per ray; N_{PI} – the number of ray-primitive intersections per ray; N_{AC} – the number of AABB constructions per ray for the ray-box culling algorithm; N_{AU}– the number of AABB updates per ray for the ray-box culling algorithm; T_R – time required for rendering; and RBC – the ray-box culling. The boldfaced values in T_R indicate the peak performance for each case. The comparisons presented in Figure 4 are based on these values. The ray-box culling method removed up to 92 percent of the ray-primitive intersection tests (Figure 6).

Scene	Leaf size	T_B (s)	N_T	N_{PI}		N_{AC}	N_{AU}	T_R(s)		$T_B + T_R$ (s)	
				no cull	RBC	RBC		no cull	RBC	no cull	RBC
Balls	1	0.031	30.66	6.25	2.77	2.21	0.12	**1.75**	1.67	**1.78**	1.64
	4	0.025	26.47	7.38	2.33	2.82	0.18	1.80	1.63	1.83	1.68
	8	0.023	24.27	9.89	2.30	2.80	0.19	1.81	**1.61**	1.83	**1.63**
	16	0.021	21.39	17.70	2.61	2.83	0.22	2.11	1.65	2.13	1.67
	32	0.019	19.09	30.52	3.14	2.69	0.23	2.73	1.71	2.75	1.73
Killeroo	1	0.28	32.39	12.10	2.42	2.83	0.46	**0.90**	0.80	**1.18**	1.08
	4	0.25	29.11	15.18	2.30	4.15	0.47	0.96	0.80	1.21	1.05
	8	0.22	25.71	16.82	2.26	3.99	0.47	0.99	**0.78**	1.21	1.00
	16	0.20	23.87	19.99	2.34	3.90	0.46	0.99	**0.78**	1.19	0.98
	32	0.18	21.51	34.53	2.76	3.78	0.46	1.23	0.79	1.41	**0.97**
Buddha	1	8.36	36.00	9.36	3.85	2.79	0.14	**1.50**	1.41	9.86	9.77
	4	7.72	31.27	12.21	3.54	3.01	0.14	1.58	1.38	9.30	9.10
	8	7.04	26.99	12.31	3.26	3.29	0.21	1.61	**1.35**	8.65	8.39
	16	6.23	25.31	15.83	3.60	3.26	0.21	1.77	1.36	8.00	7.59
	32	5.63	23.68	23.59	4.14	3.23	0.21	2.08	1.43	**7.71**	**7.06**
Sponza (direct lighting)	1	0.73	51.99	17.61	3.35	2.58	0.53	**0.38**	**0.36**	1.11	1.09
	4	0.64	48.56	21.32	3.37	4.00	0.55	0.39	0.37	1.03	1.01
	8	0.47	42.62	31.18	3.46	4.01	0.58	0.41	0.37	0.88	0.84
	16	0.33	37.96	38.40	3.95	3.68	0.65	0.41	0.37	0.74	0.70
	32	0.25	33.43	54.16	4.73	3.50	0.69	0.44	0.38	**0.69**	**0.63**
Sponza (path tracing)	1	0.73	49.84	18.70	4.20	3.16	0.52	**4.62**	4.21	**5.35**	4.94
	4	0.64	45.88	21.65	4.18	3.99	0.55	4.83	4.19	5.47	4.83
	8	0.47	41.07	28.17	4.27	3.92	0.60	5.12	4.25	5.59	4.72
	16	0.33	36.17	36.83	4.72	3.58	0.64	5.33	**4.17**	5.66	**4.50**
	32	0.25	31.32	56.83	5.62	3.33	0.68	6.62	4.32	6.87	4.57

The experimental results show that the ray-box culling algorithm achieved faster tree build time and faster ray tracing time. Applying the ray-box culling algorithm to kd-trees improved rendering performance by up to 15% (Figure 4). When we measured tree build time and ray tracing time together, the ray-box culling algorithm brought about performance improvements of up to 22%. In contrast to packet-based approaches, our method showed performance improvements irrespective of the degree of ray coherence; the proposed method showed faster ray tracing performance in the Sponza scene with both directing lighting and path tracing as compared to the case in which the ray-box culling method was not used.

A close look at the result reveals that the ray-box culling method provided a greater advantage in shallower kd-trees. Figure 5 depicts this finding. When we enabled the ray-box culling method, the difference in the rendering performance between a shallow kd-tree with a leaf size of 32 and a deep kd-tree with a leaf size of 1 was very small (Figure 5 right-top). In contrast, when the method was disabled, the shallow trees showed poor performance because many ray-primitive intersection tests were run (Figure 5 right-bottom). This feature of the ray-box culling method originated from a high culling rate of up to 92% (Figure 6). This culling rate is comparable to that of frustum culling in coherent grid traversal [18] (88%–93%) and in vertex culling [6] (90%). Furthermore, the shallow kd-trees provided a faster build time (Figure 5 left) because the number of tree nodes was reduced by larger-sized leaf nodes. Therefore, we expect that our approach can be suitable for dynamic scenes. These features are similar the features found in vertex culling [6].

Our approach requires more space for primitiveAABBs. The size of the required memory is 24 bytes per primitive. For example, if a scene is comprised of 1M primitives, our approach requires 24 MB more memory space. However, this disadvantage can be offset by using shallow trees. Figure 7 describes this feature. When we enabled the ray-box culling method, the optimal leaf size was 16. When the ray-box culling method was not used, the optimal leaf size was 1. The size of shallower kd-trees with a leaf size of 16 was approximately one-tenth of the size of the deeper kd-trees with a leaf size of 1. According to the results shown in Figure 7, the use of the ray-box culling algorithm with shallower kd-trees reduced the total memory footprint by 47%-63%.

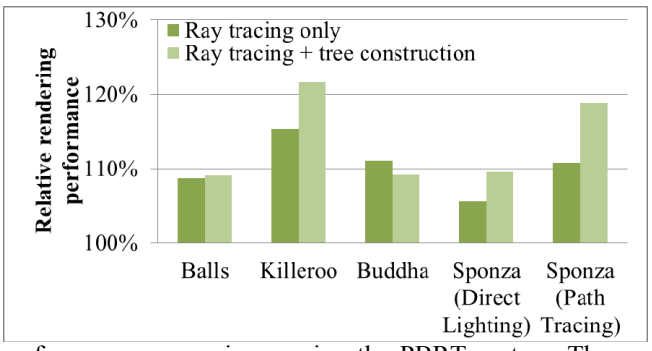


Figure 4. Peak performance comparisons using the PBRT system. The ray-box culling algorithm provided 6%-15% faster ray tracing. When tree construction time is included in the total rendering time, the performance improvement of the ray-box culling method was 8%-22%.

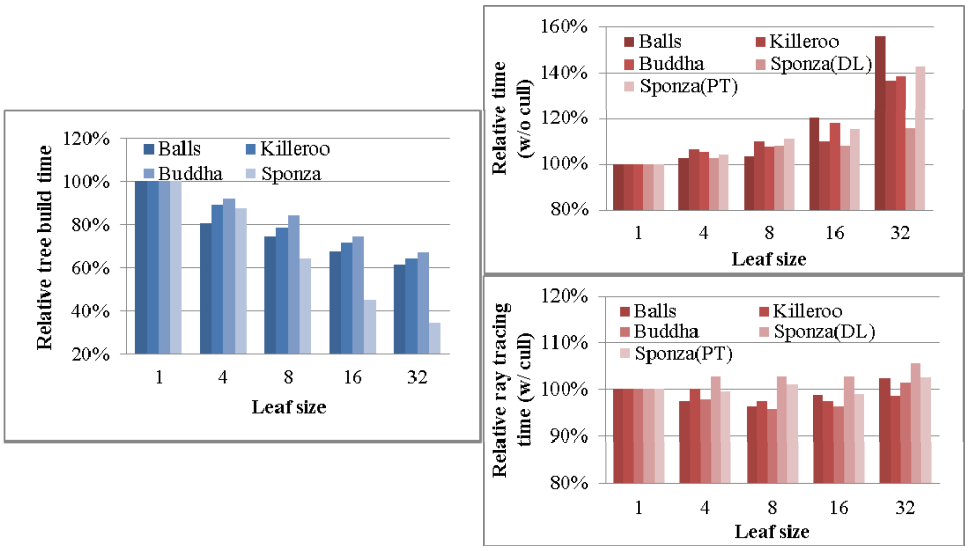


Figure 5. Relative tree build time and ray tracing time by changing leaf sizes. When larger leaf sizes were used without the ray-box culling method, tree build time decreased but ray tracing time increased. However, the ray-box culling method showed slightly better ray tracing performance when the leaf sizes were larger (up to 16). Note that DL and PT are abbreviations of direct lighting and path tracing, respectively.

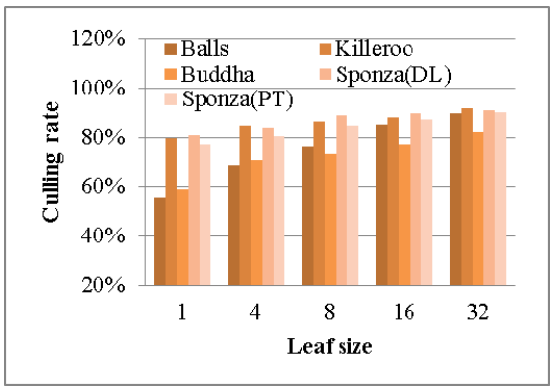
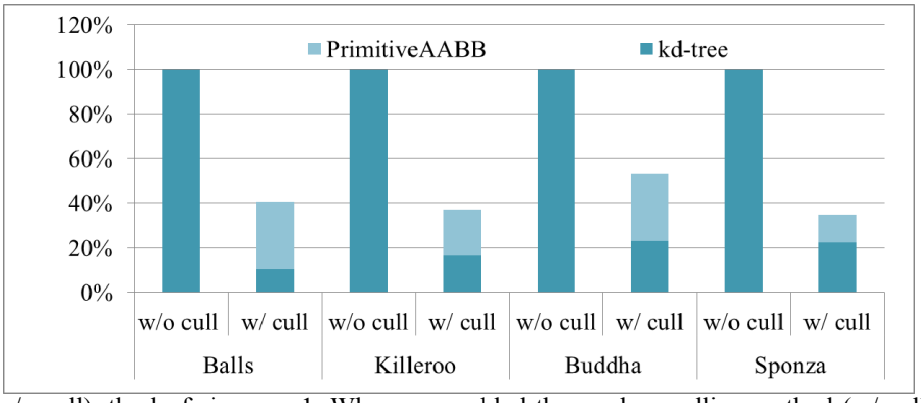


Figure 6. The culling efficiency of the ray-box culling algorithm.

Table 3. Memory footprint analysis. Each node requires 8 bytes. Each primitive list requires 4 bytes. Each PrimitiveAABB requires 24 bytes. The values presented in bold fonts represent the memory footprint at the optimal leaf size that facilitates peak performance. These values are used for comparisons in Figure 7.

Scene	Leaf size	Nodes	Primitive Lists	Memory usage of a kd-tree (KB)	Memory usage of Primitive-AABBs (KB)	Total memory footprint for the ray-box culling (KB)
Balls (7,383 primitives)	1	59567	27933	**574**	173	747
	4	16497	18498	201		374
	8	7217	12877	106		279
	16	2765	9985	60		**233**
	32	1313	9115	45		218
Killeroo (33,721 primitives)	1	313605	373997	**3910**	790	4701
	4	191393	326854	2772		3562
	8	78431	194878	1373		2164
	16	27067	112179	649		**1439**
	32	9881	75394	371		1162
Buddha (1,087,721 primitives)	1	6495245	8809544	**85156**	25493	110649
	4	4539243	8087699	67055		92548
	8	2311833	5791524	40684		66177
	16	835053	3402189	19813		**45307**
	32	304035	2280724	11284		36777
Sponza (66,564 primitives)	1	882557	1502004	**12762**	1560	14322
	4	652499	1406676	10592		12152
	8	323837	1007575	6465		8025
	16	103519	533821	2893		**4454**
	32	29821	270507	1289		2849

Figure 7. Memory footprint comparison. When we disabled the ray-box culling method (w/o cull), the leaf size was 1. When we enabled the ray-box culling method (w/ cull), the leaf size was 16. We obtained these optimal leaf sizes from the experimental results presented in Table 2.



5. CONCLUSIONS AND FUTURE WORK

This study demonstrated how to apply the ray-box culling algorithm to tree structures. Our approach showed up to 1.15× faster ray tracing performance and up to 1.22× faster total rendering performance. This speed-up was achieved by avoiding unnecessary ray-primitive intersection tests and by using shallow tree structures.

Although we have only analyzed our approach with static kd-trees, our approach offers the following benefits. First, it can be applied to any axis-aligned acceleration structure, such as kd-trees, grids, and BVHs. Second, it can be applied to any primitive types that can be bounded by an AABB, such as a triangle, a box, or a sphere, among others. Third, it can be useful for both static scenes and dynamic scenes because it efficiently reduces the number of intersection tests created by shallow trees for fast construction. Therefore, we believe that the approach in this paper can be used widely in ray tracing applications.

This paper only focused on CPU-based ray tracing. In future studies, we would like to extend our approach to GPU ray tracers. AABB/AABB overlap tests have already been utilized to accelerate collision detection routines on GPUs [17]. Therefore, we expect that our method would also be effective on GPUs.

ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for their constructive comments for this paper. We would also like to thank Jae-Hee Park for his advice.

REFERENCES

1. T. Whitted, “An improved illumination model for shaded display,” *Communications of the ACM*, Vol. 23, No. 6, 1980, pp. 343–349.
2. C. Benthin, *Realtime ray tracing on current CPU architectures*, Ph.D. thesis, Sarrland University, 2006.
3. J. Snyder and A. Barr, “Ray tracing complex models containing surface tessellations,” *ACM SIGGRAPH Computer Graphics*, Vol. 21, No.4, 1987, pp. 119–128.
4. K. Dmitriev, V. Havran, and H.-P. Seidel, “Faster ray tracing with SIMD shaft culling,” Tech. Rep. MPI-I-2004-4-006, Max-Planck Institut für Informatik, 2006.
5. S. Boulos, I. Wald, and P. Shirley, “Geometric and arithmetic culling methods for entire ray packets,” Tech. Rep. UUSCI-2006-022, SCI Institute, University of Utah, 2006.
6. A. Reshetov, “Faster ray packets - triangle intersection through vertex culling,” in *Proceedings of IEEE/EG Symposium on Interactive Ray Tracing* 2007, 2007, pp. 105–112.
7. J. Amantides and A. Woo, “A fast voxel traversal algorithm for ray tracing,” in EUROGRAPHICS ’87, 1987, pp. 3–10.
8. I. Wald, P. Slusallek, C. Benthin, and M. Wagner, “Interactive rendering with coherent ray tracing,” *Computer Graphics Forum (Proceedings of EUROGRAPHICS*

2001), Vol. 20, No. 3, 2001, pp. 153–164.
9. M. Shevtsov, A. Soupikov, and A. Kapustin, "Ray-triangle intersection algorithm for modern CPU architectures," in *Proceedings of GraphiCon 2007,* 2007, pp. 33-39.
10. I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the art in ray tracing animated scenes," *Computer Graphics Forum*, Vol. 28, No.6 , 2009, pp. 1691-1722.
11. M. Pharr and G. Humphreys, *Physically Based Rendering*, Second Edition, Morgan Kaufmann Publishers, 2010.
12. V. Havran, *Heuristic Ray Shooting Algorithms*. Ph.D. Thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000.
13. M. Ernst and G. Greiner, "Early split clipping for bounding volume hierarchies," in *Proceedings of IEEE/EG Symposium on Interactive Ray Tracing* 2007, 2007, pp. 73-78.
14. M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the Conference on High Performance Graphics* 2009, 2009, pp. 7–13.
15. I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics*, Vol. 26, No. 1, 2007.
16. J.D. MacDonald and K.S. Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, No. 3, 1990, pp. 153–166.
17. X. Zhang and Y.-J. Kim, "Interactive collision detection for deformable models using streaming AABBs," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 13, No. 2, 2007, pp. 318–329.
18. I. Wald, T. Ize, A. Kensler, A. Knoll, and S.G. Parker, "Ray tracing animated scenes using coherent grid traversal," *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, Vol. 25, No. 3, 2006, pp. 485–493.
19. A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)*, Vol. 24, No. 3, 2005, pp. 1176–1185.
20. A. Woo, "Ray tracing polygons using spatial subdivision," in *Proceedings of the conference on Graphics Interface '92*, 1992, pp. 184-191.
21. Wikipedia contributors, "Geometric primitive," Wikipedia, http://en.wikipedia.org/wiki/Geometric_primitive.

Jae-Ho Nah (羅在鎬) received B.S and M.S. degree from Department of Computer Science, Yonsei University in 2005 and 2007 respectively. Currently, he is a Ph.D student in the same department from 2007. His research interests include ray tracing, rendering algorithms, and graphics hardware.

Woo-Chan Park (朴祐贊) received M.S and Ph.D degree in Computer Science, Yonsei University in 1995 and 2000, respectively. Currently, he is a professor at the School of Computer Engineering, Sejong University, Seoul, Korea. His research interests include 3D rendering processor architecture, ray tracing accelerator, parallel rendering, high performance computer architecture, computer arithmetic, and ASIC design.

Yoon-Sig Kang (姜允植) received M.S. degree from Department of Computer Science, Yonsei University in 1999. And currently, he is a Ph.D student in the same department from 2004. He is working on dynamic ray-tracing for photon realistic image synthesis especially the acceleration structures of dynamic ray-tracing.

Tack-Don Han (韓鐸敦) is a professor in the Department of Computer Science at the Yonsei University, Korea. His research interests include high performance computer architecture, media system architecture, and wearable computing. He received Ph.D. in Computer Engineering from the University of Massachusetts.