Accelerated Entry Point Search Algorithm for Real-Time Ray-Tracing

Figure 1: *Four of the scenes used for testing purposes. From the left: "Fairy Forest" from the Utah 3D Animation Repository, Legocar from the Ompf forum model repository and Marko Dabrovic's Sponza Atrium and Sibenik Cathedral*

Abstract

Traversing an acceleration data structure, such as the Bounding Volume Hierarchy or kD-tree, takes a significant amount of the total time to render a frame in real-time ray tracing. We present a two-phase algorithm based upon MLRTA for finding deep entry points in these tree acceleration data structures in order to speed up traversal. We compare this algorithm to a base MLRTA implementation. Our results indicate an across-the-board decrease in time to find the entry point and an increase in entry point depth. The overall performance of our real-time ray-tracing system shows an increase in frames per second of up to 36% over packet-tracing and 18% over MLRTA. The improvement is algorithmic and is therefore applicable to all architectures and implementations.

CR Categories: I.3.7 [Three-Dimensional Graphics and Realism]: Raytracing, Beam Tracing.— [I.3.6]: Methodology and Techniques—Graphics data structures and data types.

Keywords: real-time ray-tracing, MLRTA, BVH, kD-tree, traversal algorithm

1 Introduction

The naïve ray-tracing algorithm involves the tracing of single rays through every object in the scene database to determine the intersection nearest to the ray origin [Appel 1968] [Whitted 1980] [Cook et al. 1984]. Modern ray-tracers use an acceleration data structure, such as the BVH or kD-tree, to reduce the candidate set for intersection from N objects to $\log N$ [Glassner 1989]. Up to 60% of total rendering time is spent traversing these acceleration data structures [Benthin 2006]. The simple tracing of single rays through an acceleration data structure, such as the kD-tree or Bounding Volume Hierarchy, which we refer to as "mono-tracing", was first improved upon by traversing multiple rays at once [Havran and Bittner 2000]. Packet-tracing [Wald et al. 2001] [Wald 2004] [Benthin 2006] is a technique that groups coherent rays (that is, rays with relatively similar directional vectors and origin point components) together to trace them through the acceleration data structure simultaneously. Highly coherent ray packets will tend to traverse the tree in the same fashion. By leveraging the SIMD capabilities of modern CPU architectures, several or all the rays in a packet can be operated on at once.

2 MLRTA

The Multi Level Ray-Tracing Algorithm (MLRTA) [Reshetov et al. 2005] further extends the concept of packets to a more general case by using a *ray proxy frustum*. This frustum is typically composed of the corner rays of a large packet. It acts as a proxy for rays that lie inside the frustum, regardless of whether or not these rays have actually been generated yet. At its simplest, the ray proxy frustum may be used for trivial rejects against the axis-aligned bounding box (AABB) of the scene geometry. We may, for example, form a ray proxy frustum that bounds a set of primary rays corresponding to a tile on the image plane. If the ray proxy frustum does not intersect the AABB, we can conclude that all rays inside the ray proxy frustum do not intersect it either.

2.1 Entry Points

It is possible that a ray proxy frustum may traverse the tree and end up wholly in a single leaf. For example, when traversing a kD-tree, the ray proxy frustum may not overlap any splitting planes. As the frustum serves as a proxy for any rays in the frustum, logically, any one of these rays traversing the kD-tree will end up in the same single leaf. Therefore, as we know the ray will end up in a specific leaf, there is no need to traverse the tree at all. We may simply intersect the ray with any objects in the leaf. We therefore say that the traversal algorithm enters the tree at that leaf node. The leaf node is our *entry point* into the tree.

The above illustrates an extreme case where the ray proxy frustum does not overlap any split planes and hence no other leaf nodes so that the entry point is at a leaf node, requiring no traversal. In a case where the ray proxy frustum overlaps two leaf nodes containing objects, both children of a common parent, the entry point is the parent node as rays inside the ray proxy frustum may terminate in either node. If one of the leaf nodes contained no objects we may safely ignore it as no intersections will occur in that leaf. The entry point is then the other leaf node.

Expanded to the general case, we define the entry point as:

"the common ancestor node in the tree of all leaves that contain objects overlapped fully or partially by the ray proxy frustum"

2.2 Entry Point Search

MLRTA is implemented using the following procedure:

- Prepare a stack data structure capable of holding kD-tree nodes and the corresponding AABB of the node in the kD-tree together in a single stack element. This is termed the bifurcation stack.
- Starting at the root node, begin traversing the kD-tree with the ray proxy frustum.
- If the ray proxy frustum must traverse both children of the current node, the current node and current AABB are pushed onto the bifurcation stack.
- The kD-tree is traversed using the ray proxy frustum until the first occupied leaf is found.
- The bifurcation stack is now frozen. No further entries may be added to it. The current leaf is marked as the current entry point candidate.
- For each node on the bifurcation stack, mark the node as a possible candidate and investigate if the tree branch not previously taken below that node contains an occupied leaf overlapped by the ray proxy frustum. If so, the possible candidate is marked as the new entry point.
- Continue until the bifurcation stack is empty.
- Return the current entry point as the entry point into the tree for all rays proxied by the frustum. The AABB stored on the bifurcation stack with the entry point is also returned for kD-tree traversal.

3 Accelerated Entry Point Search Algorithm

MLRTA's entry point search algorithm may be broken down into two phases, namely:

1. Traverse the tree with the ray frustum proxy, preparing a candidate list of entry points.
2. Investigate the candidate list, returning the best[1] entry point.

We enhance both of these phases, returning deeper entry points in phase 1 and visiting fewer nodes in phase 2. As the kD-tree acceleration data structure is a binary tree, finding an entry point one node deeper into the tree reduces the number of nodes under the entry point (assuming a complete binary tree[2]) by half, therefore in the best case halving the number of nodes visited during the traversal of the acceleration data structure by rays which the frustum proxies. Deeper entry points are also beneficial on GPUs where stacks are difficult to implement due to hardware constraints [Foley and Sugerman 2005]. As the stack size required for a full traversal from entry point to leaf is $dl - de$, where dl is the leaf depth and de the depth of the entry point, by increasing de, we lower memory requirements and the possibility that stack restarts are required when using a GPU tree traversal algorithm with a limited stack size [Horn et al. 2007].

By finding the entry point faster, we accelerate the traversal of the kD-tree, yielding more CPU time for triangle intersection and shading, ultimately culminating in an increase in renderer throughput.

3.1 Phase 1

Phase one of our algorithm prepares an entry point candidate list in a similar fashion to MLRTA. We traverse the ray proxy frustum through the kD-tree, adding nodes where both children must be traversed to the candidate list. As there is a high probability that a ray proxy frustum reaching a leaf node does not actually intersect with any object in that leaf node [Reshetov 2007], we do not freeze the candidate list until the ray proxy frustum has reached a leaf in which it actually overlaps objects stored in the leaf. In contrast, MLRTA stops when it reaches any full leaf node, regardless of whether the ray proxy frustum overlaps objects stored in that leaf or not.

Frustum Culling

In order to ascertain whether the ray proxy frustum has reached a leaf in which it overlaps an object, we employ a simple plane-based test. If all of an object's triangles are on the outer side of a plane formed by a frustum face, it is not intersected by the ray proxy frustum. A dot product is used to test if all of a triangle's vertices are on the same side of a plane. If the signs of the dot products of each vertex are the same then the triangle does not overlap the plane. Using SIMD, we are able to concurrently test all four planes of the ray proxy frustum. The normals of the frustum planes are already pre-calculated in order to cull kD-tree nodes and thus there is little overhead to this test. This phase is similar to the shaft culling techniques presented in [Dmitriev et al. 2004].

3.2 Phase 2

The candidate list contains the nodes on a traversal from root to overlapped leaf where the ray proxy frustum possibly overlaps both child sub-trees of that node. The candidate nodes are therefore ordered by depth. Candidate nodes at lower levels exist in sub-trees of higher nodes. Therefore, if we can ascertain that both of a candidate entry point's sub-trees contain leaves overlapped by the ray proxy frustum, we know that any entry point below the current entry point will not encompass all of the sub-trees of the current candidate. We can therefore cull all entry points in a candidate list at a tree depth below any point found with both sub-trees containing leaves with overlapped objects .

As investigating each potential entry point involves a traversal from that point to an occupied leaf, by not having to test every entry point we greatly decrease the nodes traversed and therefore the time required to perform such traversals. As the Accelerated Entry Point Search Algorithm (AEPSA) performs tests in a top-down manner from the highest potential entry point, when it is known that an entry point cannot be deeper in the tree than the current point, we may reject any nodes remaining in the list. MLRTA performs a bottom-up test of entry point candidates and therefore requires each candidate node be tested in turn.

[1] We define the best entry point as the node which the minimum number of traversal steps are necessary for all in the rays in the proxy frustum to reach a leaf node containing objects.

[2] A binary tree in which all leaf nodes are at the same depth.

The candidate list/bifurcation stack is populated by a root to leaf traversal of a tree with n nodes, therefore it will contain at least one node (the leaf the traversal terminates in) and at most $\log n$ nodes (each node visited in the traversal, if the ray proxy frustum overlaps both child nodes). The number of possible entry points in the list/stack can therefore be written as $p(\log n)$ where p is a "branching factor" and $0 < p \leq 1$.

During phase 2, in a candidate list/stack with k entries, MLRTA will visit $k(\log n)$ nodes. This is because each entry in the stack requires a traversal from that candidate entry point to a leaf. AEPSA will visit $x(\log n)$, where $x \leq k$ as AEPSA can exit early as soon as it encounters a candidate with a child with an overlapped, occupied leaf. The bounds of x are $1 \leq x \leq \log n$. We can therefore prove that AEPSA will at worst spend the same time as MLRTA searching for the entry point and at no point will it spend longer.

By using this entry point test procedure, we yield the same entry points as MLRTA when considering the same candidate list but without requiring the entire candidate list search.

3.3 Algorithm Outline

AEPSA is implemented using the following procedure:

- Prepare a queue data structure capable of holding AABBs and tree nodes together in a single stack element. This is termed the entry point candidate queue.

- Starting at the root node, begin traversing the kD-tree with the ray proxy frustum.

- If the ray frustum must traverse both children of the current node, the current node and current AABB are added to the entry point candidate queue.

- The kD-tree is traversed using the ray proxy frustum until the first occupied leaf is found that contains objects intersected by the ray proxy frustum

- The entry point candidate queue is now frozen. No further entries may be added to it.

- Take the first candidate from the queue and set it as the candidate entry point. Investigate if the kD-tree branch not previously taken below that candidate entry point node contains an occupied leaf overlapped by the ray proxy frustum. If so, return the current entry point. The AABB stored with the entry point is also returned for kD-tree traversal.

- If necessary, continue until the queue is empty.

Figure 2 illustrates the full algorithm and compares it with a MLRTA entry point search into the same kD-tree. A traversal by MLRTA from root to the first occupied leaf in this instance will yield the bifurcation stack [2,1,0] (Node 2 being at the top of the stack and 0 at the bottom). After testing node 2, 1 and 0, MLRTA will return 0 as the entry point into the tree as an occupied leaf 9 is also overlapped by the ray proxy frustum. AEPSA on its search for the first occupied leaf containing triangles overlapping the ray proxy frustum will yield the candidate queue [9]. As this is the only candidate in the queue, we return 9 as the entry point. AEPSA in this case has produced an entry point 3 levels deeper into the tree and instead of entering all rays proxied by the frustum at the root, enters them at a leaf node meaning that no traversal is required. See Appendix A for a pseudo-code implementation of AEPSA.

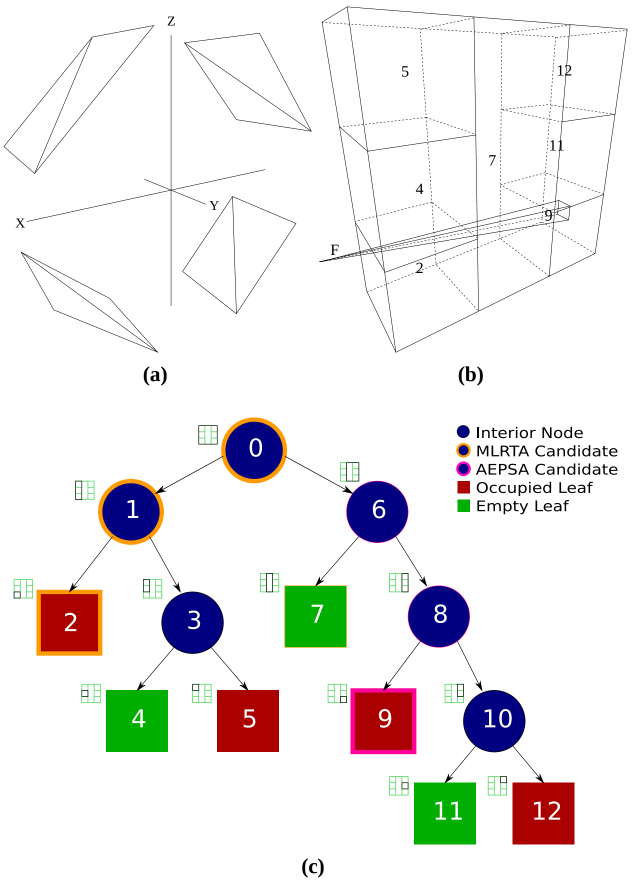
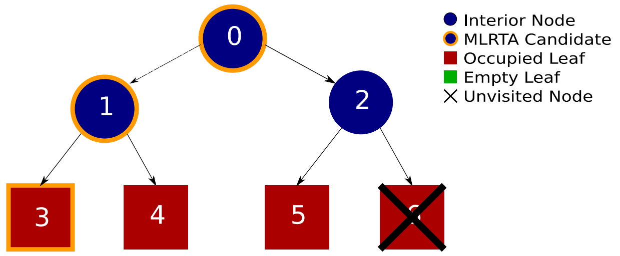


Figure 2: *(a) A simple scene formed by eight triangles inclined at a 45° angle to the XY plane. (b) A visualisation of the leaf nodes formed by a kD-tree compiler using a termination criterion of a maximum of 2 triangles per leaf. Also shown is an example ray proxy frustum that enters the scene from the left, penetrating leaf node 2, but missing the triangles in the leaf. The ray proxy frustum continues on, finally penetrating two triangles in leaf node 9. (c) A layout of the kD-tree compiled in (b).*

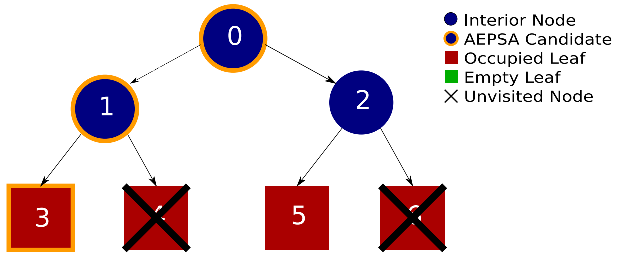
4 Comparison with MLRTA

We begin our comparisons of AEPSA to MLRTA by considering two extreme cases. We compare the number of nodes visited by both.

Case one (see Figure 3) consists of a complete balanced kD-tree containing N nodes in which all leaf nodes contain objects. The ray proxy frustum fully overlaps the entire tree and therefore every object in all leaves. The common ancestor of all intersected occupied leaves is then the root node. The traversal from root to the first occupied leaf adds $\log N$ candidates to the bifurcation stack and in the case of AEPSA, the entry point candidate queue. To check each candidate entry point, a traversal of the tree from the candidate node to a leaf is performed. Each traversal will visit $\log N$ nodes. As MLRTA will perform a traversal for each node on the bifurcation stack, the number of visited nodes is N. Given that each leaf is fully overlapped by the ray proxy frustum, during phase 2 of AEPSA, a fully overlapped node will be found testing the first



(a)



(b)

Figure 3: *A comparison of the visited nodes in a tree where the ray proxy frustum fully overlaps each leaf. (a) MLRTA: a traversal from the root node 0 to the first occupied leaf 3 adds the nodes [3,1,0] to the bifurcation stack. 3 is popped and marked as a potential entry point. Node 1 is then popped and investigated. As a traversal from 1 to 4 finds an occupied leaf, the candidate entry point is now 1. Node 0 is then popped. A traversal from node 0 passes through node 2 to an occupied leaf at 5. Node 0 is then marked as the candidate entry point. As the stack is now empty the current entry point 0 is returned. (b) AESPA: a traversal from the root node 0 to the first occupied overlapped leaf 3 adds the nodes [0,1,3] to the candidate queue. The first entry (and highest in the tree) 0 is investigated and a traversal from 0 to leaf 5 yields an occupied overlapped leaf. Node 0 is returned as the entry point.*

candidate (the tree root node) in the candidate queue. At this point, AEPSA will return the root node as the entry point after a single $\log N$ traversal.

Case two (see Figure 4) consists of a complete balanced kD-tree containing N nodes, in which 50% of all leaf nodes contain objects. The ray proxy frustum overlaps two empty leaf nodes and a single occupied leaf node. The common ancestor of all intersected occupied leaves is therefore the leaf node itself. MLRTA performs a traversal from the root node 0 to the first occupied leaf 3 and adds the nodes [3,1,0] to the bifurcation stack. Node 3 is popped and marked as a potential entry point. Node 1 is then popped and investigated. As a traversal from 1 to 3 finds an empty leaf, the candidate entry point is still leaf 3. Node 0 is then popped and investigated. A traversal from the candidate 0 along the right sub-tree visits in order, the nodes 2, 5 and 6. As leaf 5 is empty and leaf 6 is not overlapped, the candidate entry point is still leaf node 3. The bifurcation node is now empty, therefore node 3 is returned as the entry point. AEPSA performs a traversal from the root node 0 to the first occupied overlapped leaf 3 and adds the nodes [0,1,3] to the candidate queue. The first entry (and highest in the tree) 0 is investigated and traverses in order, the nodes 2, 5 and 6. As there is no overlap, AEPSA has not yet found an entry point. Node 1 is then taken from the queue. As node 4 is empty, AEPSA has not yet found the entry point. The final node 3 is taken from the queue. As it is the last entry in the queue and is occupied, it is by definition overlapped by the ray proxy frustum and is returned as the entry point

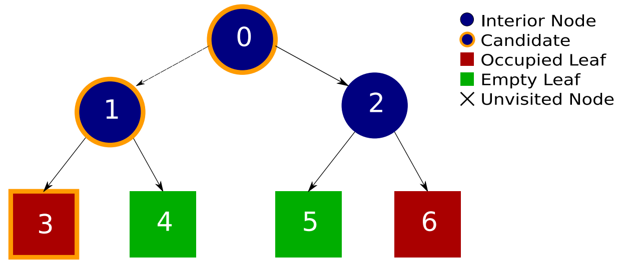


Figure 4: *A comparison of the visited nodes in a tree where the ray proxy frustum fully overlaps leaves 3, 4 and 5. Leaf 6 is not overlapped by the ray proxy frustum.*

The traversal from root to the first occupied leaf (also the only overlapped leaf) adds $\log N$ candidates to the bifurcation stack or, in the case of AEPSA, the entry point candidate queue. MLRTA will mark the leaf as a potential entry point and test all the leaves above it in the leaf again yielding $\log N \log N$ nodes visited. As AEPSA starts testing at the highest node in the list, it will need to test all entries in the list before it reaches the lowest candidate which is the leaf node that is the correct entry point. AEPSA therefore in this case visits the same number of nodes as MLRTA as it can not exit early.

5 Evaluation

We collect the following data on a per-scene basis for both MLRTA and AEPSA:

- The average depth of the entry point in the tree
- The average time to find the entry point
- The average number of nodes visited by rays entering the tree at the found entry point

These averages are calculated from all ray proxy frustums used in the scene. In the event that a ray proxy frustum penetrates the scene fully without intersecting any objects, we count this ray proxy frustum as having entered the scene at the average depth.

5.1 Implementation Details

Our real-time ray-tracer employs SIMD packet tracing [Wald et al. 2001][Wald 2004] of kD-trees. Incoherent packets are traced using an omni-directional traversal algorithm [Reshetov 2006]. We employ a fast $O(N \log N)$ kD-tree compiler [Havran 2005] [Benthin 2006] biased towards the early cutoff of empty volumes of space [Hurley et al. 2002]. kD-Tree build termination is based on the well-known SAH cost metric [MacDonald and Booth 1990] [Havran 2000]. Our MLRTA implementation is based on work presented in [Benthin 2006] and [Reshetov et al. 2005].

All results are generated on a dual Intel Xeon E5335 at 2.00GHz. We render to a 512 x 512 viewport with a single light source located at the eye-point. For timing purposes, we use the cycle-accurate RDTSC instruction [Intel 2006] on Intel's x86 Core 2 Architecture.

5.2 Test Scenes

In order to fully test AEPSA, our test suite consists of 14 scenes with discrete triangle counts ranging from 240 to over 1 million. The scenes differ in complexity and form. Several of them (the

Scene	Tris	Leaves	%Empty	AvgDepth
Sponza	79076	137427	35.57	21.62
Buddha	1087716	78344	43.37	20.72
Jagd	69399	107030	27.19	21.81
Dw Truck	125691	90734	33.62	21.39
Legocar	10882	25319	32.78	20.01
Sculpture	50772	84840	37.36	21.93
Dragon	849890	129048	46.11	22.4
Deo10k	20000	86026	30.81	21.14
Bunny	69452	156525	37.91	24.03
Kitchen	181745	130265	36.14	23.15
Room	240	620	36.13	11.24
FairyForest	174118	111419	37.75	23.45
Scene6	805	2588	28.44	16.94
Sibenik	76651	114724	34.84	22.81

Table 1: *kD-tree compiler statistics for our test scene database. From left to right, the scene name, number of triangles in the geometry, number of leaves, percentage of leaves that are empty in the final tree and average depth of all leaves are given.*

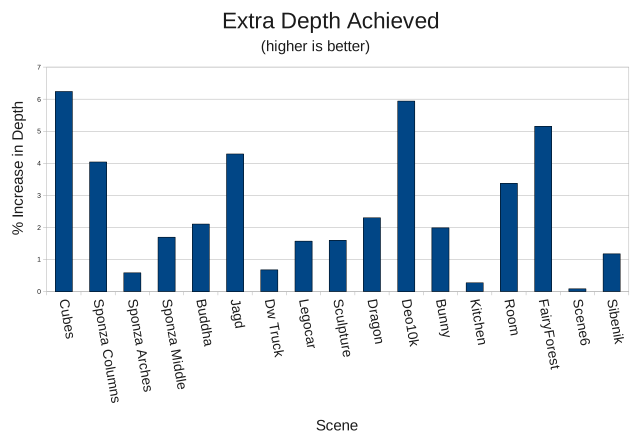


Figure 5: *Percentage of extra depth achieved over MLRTA. That is, $(Ad - Md)/Md * 100$, where Ad and Md are the average depths of the entry points in the kD-Tree returned by the MLRTA and AEPSA algorithms, respectively.*

Stanford Models [Stanford]) are the output of laser scanning and contain mostly non-axis aligned triangles of a relatively similar size. Others are architectural models exhibiting the opposite characteristics. Four of these scenes are illustrated in Figure 1. In addition to this, we provide statistics for the kD-trees generated from these scenes in Table 1.

6 Results

We will now discuss the results obtained using our method.

6.1 Phase 1

Phase 1 (preparation of the entry point candidate list) is scored on its ability to generate candidate lists containing deeper entry points than previously found. All of our test scenes present a moderate average increase in depth (see Figure 5). It is important to remember though that a depth increase of one level in the tree has to the potential to reduce the number of nodes under the entry point by half. Mean extra depth achieved was 2.54% with a standard deviation of 1.96%. Maximum extra depth achieved was 6.24%.

6.2 Phase 2

Phase 2 (returning the best entry point in the candidate list) is scored on its speed to find the entry point. Our results (see Figure 6) indicate speedups up to 144%, with a mean speedup of 57.68% and a standard deviation of 40.75%. No scene exhibits a slowdown, as predicted in Section 3.2. This is a result of the decreased number of nodes visited due to not needing to scan the entire candidate list.

6.3 Overall Performance

In order to test the overall performance of our new algorithm, we measure the frames per second achieved by our real-time ray-tracer using basic packet-tracing, MLRTA and AEPSA across our test scene database. Results show an across-the-board gain in rendering speed using AEPSA of up to 36% over packet-tracing and a speedup of up to 18% over MLRTA (see Figure 7).

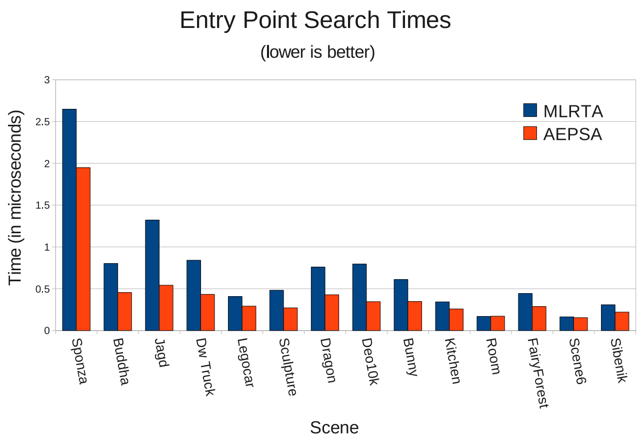


Figure 6: *Speedups achieved over MLRTA in finding the entry point in phase 2. In order to compare the second phase times more accurately, we use MLRTA phase 1 for both algorithms. This ensures that both algorithms have the same number of candidate nodes to check and that both algorithms return the same entry point.*

7 Discussion

Our tests indicate that in certain cases MLRTA is detrimental to rendering speeds. Two scenes, "Jagd" and "DW Truck" exhibit slowdowns under MLRTA. AEPSA shows slowdowns on neither of these scenes. The entry point search time (see Figure 6) for both of these scenes under AEPSA is on the order of one half the time MLRTA takes, indicating that too long of an EP search time can outweigh any benefits gained. In all cases the performance of AEPSA is greater than or equal to the rendering speed exhibited by MLRTA.

Using a Pearson Product-Moment Correlation we find no significant simple linear correlation between the tree characteristics in table 1 and overall speedup, indicating that if such a correlation exists, it may be a non-linear function of one or more variables.

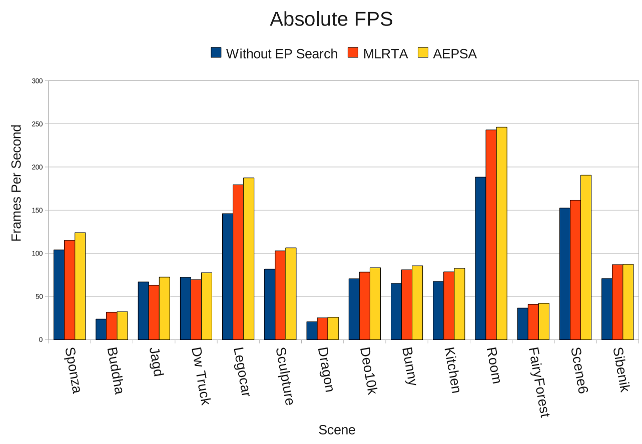


Figure 7: *Using basic packet tracing as a baseline, we compare the overall performance of our real-time ray-tracer using AEPSA and MLRTA. In all cases AEPSA produces a speedup.*

8 Further Work

As we use culling techniques in our leaf nodes during the entry-point search, it may be beneficial to skew the kD-tree compiler towards creating leaves of larger volume. Such a skewing may make it easier to cull leaf nodes as the ray proxy frustum will be more likely to pass through leaves without intersecting any geometry stored in the node. We intend to investigate tuning tree creation to leverage our algorithm's strengths.

Entry point search algorithms have been used with other acceleration data structures [Wald et al. 2006]. We intend to investigate the use of AEPSA with bounding volume hierarchies and the bounding interval hierarchy [Wächter and Keller 2006] [Waechter 2007].

9 Conclusion

We have presented a two-phase extension to MLRTA for finding deep entry points in acceleration data structures such as kD-trees or BVHs for real-time ray-tracing. We have compared this algorithm to the state-of-the-art entry point search algorithm on which we base our work. Our results indicate a decrease in time to find the entry point and an increase in entry point depth across all of our tested scenes. The overall performance of our real-time ray-tracing system showed an increase in frames per second of up to 36%.

10 Acknowledgements

This section purposely withheld for review purposes.

A AEPSA pseudo-code

The following is a simplified implementation using a recursive function to find the candidates. Our actual implemetions of ML-RTA and AEPSA are completely iterative functions with software stacks for performance.

```
PROC aepsa(tree, frustum)
  stack //holds candidates
  find_candidates(root(tree),
                  frustum, stack)

  WHILE NOT empty(stack) DO
    node = pop(stack)
    IF traverse_to_leaf(frustum, n)
      along path to leaf not taken
      overlaps non-empty leaf THEN
      RETURN WITH n
    ENDIF
  ENDWHILE

  RETURN WITH NULL
ENDPROC


PROC find_candidates(node, frustum, stack)
  IF node IS leaf THEN
    i = intersect(frustum, leaf);
    IF i == TRUE THEN
      stack.push(node);
      RETURN WITH i;
    ENDIF
  ENDIF

  s = find_candidates(left(node)
    OR find_candidates(right(node));

  IF s == TRUE THEN
    push(stack, node)
  ENDIF
ENDPROC
```

References

APPEL, A. 1968. Some Techniques for Shading Machine Renderings of Solids. In *AFIPS 1968 Spring Joint Computer Conf.*, vol. 32, 37–45.

BENTHIN, C. 2006. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Computer Graphics Group, Saarland University.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, vol. 18, 137–45.

DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. 2004. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut fr Informatik, Saarbrücken, Germany, December.

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree Acceleration Structures for a GPU Raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM Press, New York, NY, USA, 15–22.

GLASSNER, A. S., Ed. 1989. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK.

HAVRAN, V., AND BITTNER, J. 2000. Lcts: Ray shooting using longest common traversal sequences. In *Proceedings of Eurographics (EG'00)*, 59–70.

HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

HAVRAN, V., 2005. On The Kd-tree Construction Algorithms with Surface Area Heuristics. http://ompf.org/forum/viewtopic.php?t=19.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d Tree GPU Raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM Press, New York, NY, USA, 167–174.

HURLEY, J., KAPUSTIN, A., RESHETOV, A., AND SOUPIKOV, A. 2002. Fast Ray Tracing for Modern General Purpose CPU. In *Proceedings of GraphiCon 2002*.

INTEL. 2006. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*. ch. 4, 246–247.

MACDONALD, D. J., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *Vis. Comput. 6*, 3, 153–166.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM Press, New York, NY, USA, 1176–1185.

RESHETOV, A. 2006. Omnidirectional ray tracing traversal algorithm for kd-trees. In *In Proceedings of the IEEE Symposium on Interactive Ray Tracing, pages*, 57–60.

RESHETOV, A. 2007. Faster ray packets - triangle intersection through vertex culling. In *SIGGRAPH '07: ACM SIGGRAPH 2007 posters*, ACM, New York, NY, USA, 171.

STANFORD. The Stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep.

WÄCHTER, C., AND KELLER, A. 2006. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 (Proc. of 17th Eurographics Symposium on Rendering)*, T. Akenine-Möller and W. Heidrich, Eds., 139–149.

WAECHTER, C. 2007. *Quasi-Monte Carlo light transport simulation by efficient ray tracing*. Ph.d. thesis, University of Ulm.

WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, Blackwell Publishers, Oxford, A. Chalmers and T.-M. Rhyne, Eds., vol. 20, 153–164. available at graphics.cs.uni-sb.de/ wald/Publications.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM Press, New York, NY, USA, 485–493.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June), 343–349.