

Multi-level tetrahedralization-based accelerator for ray-tracing animated scenes

Aytek Aman¹ | Serkan Demirci¹ | Uğur Gudukbay¹ | Ingo Wald²

¹Department of Computer Engineering, Bilkent University, Ankara, Turkey

²NVIDIA, Salt Lake City, Utah, USA

Correspondence

Uğur Gudukbay, Department of Computer Engineering, Bilkent University, Ankara, Turkey.
Email: gudukbay@cs.bilkent.edu.tr

Funding information

The Scientific and Technological Research Council of Turkey (TUBITAK), Grant/Award Number: 117E881

Abstract

We describe a hybrid acceleration structure for ray tracing. The hybrid structure is a Bounding Volume Hierarchy (BVH) where the leaf nodes are tetrahedralized for a decent ray-surface intersection performance. We use the hybrid acceleration structure (BTH) in a two-level acceleration structure for rendering animated scenes. There is a BVH at the top level in this two-level structure and the proposed hybrid structure (BTH) at the bottom level. We test the proposed two-level structure (BVH-BTH) for various animated scenes and obtained promising results against other acceleration structures in terms of rendering times. The two-level BVH-BTH structure outperforms the two-level BVH structure for the tested dynamic scenes.

KEYWORDS

acceleration structure, Bounding Volume Hierarchy, *k*-d tree, ray tracing, tetrahedralization

1 | INTRODUCTION

A significant amount of computational time is dedicated to finding the nearest surface that a ray hits in ray tracing. With the introduction of path tracing¹ for global illumination, the importance of fast nearest hit tests increased. Path tracing is a form of ray tracing that uses the Monte Carlo method to generate realistic images. Path tracing requires a large number of lightpaths to be simulated. The quality of the rendered image in path tracing depends directly on the number of simulated lightpaths.

Given a scene and a ray, a naive way to find the nearest hit requires $\Theta(N)$ computational cost, where N is the number of primitives in the scene. This computational cost is high for most of the ray-tracing algorithms. To speed up ray-surface intersection tests, various spatial acceleration structures, such as regular grids, octrees, *k*-d trees, bounding volume hierarchies, are proposed in the literature. Such spatial acceleration structures reduce the number of ray-surface intersection tests by eliminating some of the candidate primitives. In ray tracing, one can achieve substantial gains by using such acceleration structures.

One alternative spatial acceleration structure is tetrahedralizations. Lagae and Dutré² proposed the use of constrained tetrahedralizations as spatial acceleration structure. Aman et al.³ showed that tetrahedralizations perform better than the other structures for near-hits. Although constrained tetrahedralizations are alternatives to well-known acceleration structures, tetrahedralization algorithms' shape quality requirements like minimum dihedral angle and non-self-intersecting geometry make their usage limited in real-world ray-tracing applications. We propose a BVH-Tetrahedral mesh hybrid acceleration structure (BTH), which combines the strengths of BVH and tetrahedralizations. The main contributions are as follows.

1. We propose a hybrid acceleration structure, BTH, composed of a BVH and tetrahedralizations at the leaf nodes. We describe how we construct and traverse the proposed hybrid structure.
2. We propose methods to find an approximate nearest-hit cost for the tetrahedralization acceleration structure.
3. We show how we can adapt the hybrid structure to dynamic scenes.

The organization of the paper is as follows. Section 2 summarizes the existing acceleration structures, their strengths, and weaknesses. Section 3 introduces construction and nearest-hit traversal algorithms for the BTH acceleration structure. We also describe the nearest-hit cost approximation methods for the tetrahedralization acceleration structure used by the BTH construction algorithm. Section 4 shows how we can adapt the BTH structure for dynamic scenes. Section 5 presents the experimental setup and results of the experiments. Section 6 provides conclusions and future research directions.

2 | BACKGROUND AND RELATED WORK

2.1 | Regular grids

One of the basic nearest hit acceleration structures is uniform grids. Uniform grids⁴ subdivide the scene space into uniform-sized cells. Each cell stores a list of primitives that occupy the cell. Multiple cells can contain the same primitive if the primitive spans various cells. The nearest-hit traversal algorithm on grids, often referred to as the three-dimensional digital differential analyzer (3D-DDA) algorithm, visits each cell along the ray. As each cell is traversed, all of the primitives in the cell's list are tested for intersections. The traversal stops when a cell contains a primitive that intersects the ray. Since cells are tested in ray's order, starting from the cell that ray's origin resides in, along the ray's direction. Clearly and Wyvill⁵ showed that for scenes with N evenly distributed small, equally sized primitives, optimal case is to subdivide each axis into $\sqrt[3]{N}$ cells, with minimum average $\Theta(\sqrt[3]{N})$ time and $\mathcal{O}(N)$ space complexity.

2.2 | k -d trees

In k -d trees,⁶ a scene is recursively subdivided into two parts in such a way that the number of primitives on each side of the dividing plane is more or less equal. A primitive is associated with a half-space if the half-space contains a part of the primitive. If a primitive lies in both half-spaces, it is included in both sub-trees. The nearest-hit traversal in k -d trees starts from the root. In each split, half-spaces are tested for intersection in the ray's traversal order. Traversal descends through the tree until a terminal node is reached. If a terminal node is encountered, each primitive inside the terminal node is tested for intersection. The intersection nearest to the ray's origin is reported.

Surface area heuristic (SAH)^{7,8} is the most commonly used heuristic for the construction of k -d trees. SAH estimates the expected cost of the nearest-hit test for a long uniformly distributed random ray. Given a split s , we calculate the SAH cost as

$$Cost_{SAH}(s) = \frac{C_i}{SA(s)}(n_l SA(l) + n_r SA(r)) + C_t, \quad (1)$$

where the SA function calculates the node's bounding box's surface area. C_i constant is the cost of the ray-primitive intersection, C_t constant is the cost of traversing a node. n_l and n_r are the numbers of primitives in the left and right half-spaces. The SAH-based k -d tree construction further improved in.⁹

2.3 | Bounding Volume Hierarchies

One famous spatial acceleration structure is Bounding Volume Hierarchy (BVH). BVHs are composed of a hierarchy of partitions. Each partition is represented by a volume that encloses all primitives in that partition. Unlike other acceleration structures, BVHs partition primitives instead of subdividing the scene space; hence, bounding volumes can intersect. Because BVHs partition primitives, primitives are not duplicated in BVHs, unlike k -d trees. Bounding volumes in the

hierarchy provide a simple method for finding the nearest hit. If a ray does not pass through a bounding volume, primitives represented by this bounding volume are not tested. Therefore, the nearest-hit algorithm recursively traverses the hierarchy in depth-first order. If the algorithm encounters an inner node, the algorithm checks whether a ray passes through the bounding volume. If the ray does not pass through the bounding volume, the algorithm does not traverse its children. If the algorithm encounters a leaf node, it checks each primitive with the ray for an intersection. Since BVHs bounding volumes can intersect, the algorithm does not stop when the first intersection is found. The algorithm needs to test all possible nodes before finding the nearest hit.

Goldsmith and Salmon⁷ proposed SAH and used incremental insertions for construction. Incremental construction using SAH leads to poor quality BVHs. Müller and Fellner¹⁰ used SAH to build a BVH in a top-down manner. Wald et al.¹¹ used centroid-based SAH partitioning to improve top-down construction with $\Theta(N \log^2(N))$ average construction time. Streamed binning idea from *k*-d trees¹² are applied to BVHs by Wald¹³ to improve the runtime of the construction algorithm.

2.4 | Tetrahedralizations

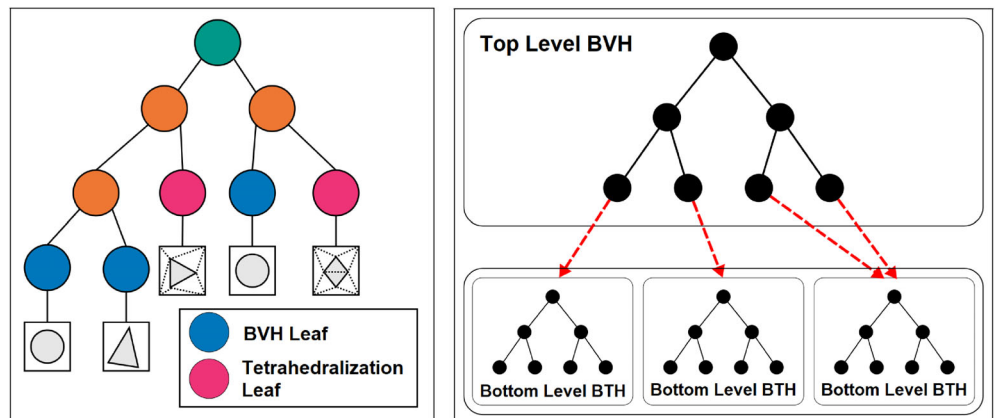
Tetrahedralizations are 3D equivalents of triangulations in a plane. Similarly, constrained tetrahedralization of a set of points, line segments, and faces is a tetrahedralization that conforms to given constraints. Many tetrahedralization algorithms assume that input constraints are formalized as Piecewise Linear Complex (PLC).¹⁴ Recently, Hu et al.'s TetWild¹⁵ tetrahedralization algorithm can generate tetrahedral meshes from triangle soup. Although their approach accepts any kind of geometry that a triangle soup can represent, it generates an approximate tetrahedralization of the input geometry.

In the ray-tracing context, the constrained tetrahedralization of the scene is used as an acceleration structure. Faces in the scene are considered constrained faces in tetrahedralization. Nearest-hit tests in the tetrahedralization structure are similar to grids. Starting from the source tetrahedron, a tetrahedron containing the ray's origin, each tetrahedron is traversed in order using shared faces, one tetrahedron at a time. Unlike grids, instead of storing input faces inside cells, in constrained tetrahedralization, input faces are stored as some of the tetrahedra's faces. The traversal ends when a ray hits such a face, and the face is reported as the nearest hit. To determine the next tetrahedron in traversal, Lagae and Dutré² used the scalar triple product method. Maria et al.¹⁶ used Plücker coordinates for determining the next tetrahedron to improve traversal efficiency. Aman et al.³ proposed tetrahedralization representations for ray tracing. They proposed *Tet32*, *Tet20*, and *Tet16* tetrahedron storage schemes, along with nearest-hit traversal algorithms. Their traversal method uses the projected 2D ray's coordinates to traverse a tetrahedralization efficiently.

3 | BOUNDING VOLUME HIERARCHY-TETRAHEDRALIZATION HYBRID STRUCTURE

The Bounding Volume Hierarchy-Tetrahedralization Hybrid (BTH) acceleration structure is composed of a BVH where some leaves of the hierarchy contain tetrahedralizations of the primitives instead of primitive lists (see Figure 1 (a)).

FIGURE 1 The proposed BVH-based hybrid acceleration structures.
 (a) BVH-Tetrahedralization hybrid (BTH) structure.
 (b) Two-Level BVH-BTH structure



An axis-aligned bounding box encloses each tetrahedralization leaf node. Traversal in BTH structure is similar to BVH traversal. We recursively traverse each node until we reach a leaf node. In the leaf node, if we have a tetrahedralization, we traverse the tetrahedralization. Otherwise, we check each primitive in the leaf for an intersection. Since the faces visited in tetrahedralizations are sorted according to the ray direction, we can exit the traversal early once we find a hit in a tetrahedralization. BTH combines the strengths of BVH and tetrahedralization. We can construct BTH for the scenes where scenes cannot be completely tetrahedralized. In BTH, a part of the geometry to be tetrahedralized can be selected and used as a leaf node. The construction of BTH is faster than that of tetrahedralizations due to the non-linear complexity of tetrahedralizations.

3.1 | Construction

We build a BVH using the existing SAH-based methods. Then, we transform the built BVH structure into a BTH structure. We first construct a complete BVH for the given geometry. Then we select some nodes of the BVH structure to be a tetrahedralization structure. We trim the selected nodes and construct tetrahedralizations for the nodes. We explain the three steps of the top-down construction algorithm (see Procedure CONSTRUCTBTHTOPDOWN in Figure 2) as follows:

First Step: We use the binned BVH construction method¹³ to construct a BVH in a top-down fashion. We use the SAH heuristic to build the BVH for the scene.

Second Step: We mark the suitable nodes that can be tetrahedralized. We choose not to tetrahedralize the geometries with self-intersecting primitives as it is costly. In this step, we mark the BVH nodes that do not contain self-intersecting primitives. We call such nodes as *suitable nodes*. We mark these nodes by a recursive algorithm that

<pre> 1: procedure CONSTRUCTBTHTOPDOWN(<i>primitives</i>) 2: <i>bvh</i> ← CONSTRUCTBVHTOPDOWN(<i>primitives</i>) 3: MARKSUITABLENODES(<i>bvh</i>) 4: <i>bth</i> ← CONSTRUCTTETRAHEDRALIZATIONS(<i>bvh</i>) 5: return <i>bth</i> 6: end procedure </pre>	
<pre> 1: procedure MARKSUITABLENODES(<i>node</i>) 2: if <i>node</i> is a leaf node then 3: if <i>node.primitives</i> has no self intersection then 4: MARKSUITABLE(<i>node</i>) 5: end if 6: else 7: <i>l</i> ← <i>node.left_child</i> 8: <i>r</i> ← <i>node.right_child</i> 9: MARKSUITABLENODES(<i>l</i>) 10: MARKSUITABLENODES(<i>r</i>) 11: if <i>l</i> and <i>r</i> are suitable then 12: if <i>l.primitives</i> and <i>r.primitives</i> not intersect then 13: MARKSUITABLE(<i>node</i>) 14: end if 15: end if 16: end if 17: end procedure </pre>	<pre> 1: procedure BVHCOLLISION(<i>node</i>₁, <i>node</i>₂) 2: if <i>node</i>₁.<i>bounding_volume</i> and <i>node</i>₂.<i>bounding_volume</i> do not overlap then 3: return 4: end if 5: if <i>node</i>₁ and <i>node</i>₂ are leaf nodes then 6: CHECKINTERSECTION(<i>node</i>₁, <i>node</i>₂) 7: else 8: if descend <i>node</i>₁ then 9: BVHCOLLISION(<i>node</i>₁.<i>left_child</i>, <i>node</i>₂) 10: BVHCOLLISION(<i>node</i>₁.<i>right_child</i>, <i>node</i>₂) 11: else 12: BVHCOLLISION(<i>node</i>₁, <i>node</i>₂.<i>left_child</i>) 13: BVHCOLLISION(<i>node</i>₁, <i>node</i>₂.<i>right_child</i>) 14: end if 15: end if 16: end procedure </pre>
<pre> 1: procedure CONSTRUCTTETRAHEDRA(<i>node</i>) 2: if <i>node</i> is marked then 3: if <i>Cost</i>_{tet}(<i>node</i>) < <i>Cost</i>_{SAH}(<i>node</i>) then 4: PRUNE(<i>node</i>) 5: TETRAHEDRALIZE(<i>node.primitives</i>) 6: return 7: end if 8: end if 9: if <i>node</i> is not a leaf node then 10: CONSTRUCTTETRAHEDRA(<i>node.left_child</i>) 11: CONSTRUCTTETRAHEDRA(<i>node.right_child</i>) 12: end if 13: end procedure </pre>	<pre> 1: procedure BTHINTERSECT(<i>node</i>, <i>ray</i>) 2: if <i>node</i> is a leaf then 3: if <i>node</i> is a tetrahedralization then 4: τ_0 ← Find first tetrahedron 5: TETRAHEDRAINTERSECT(<i>ray</i>, <i>node</i>, τ_0) 6: else 7: for all <i>primitive</i> in <i>node</i> do 8: INTERSECT(<i>primitive</i>, <i>ray</i>) 9: end for 10: end if 11: else 12: BTHINTERSECT(<i>node.left</i>) 13: BTHINTERSECT(<i>node.right</i>) 14: end if 15: end procedure </pre>

FIGURE 2 Algorithms for the construction and traversal of the hybrid acceleration structure

visits each node and checks for self-intersection (see Procedure MARKSUITABLENODES in Figure 2). The algorithm marks the non-self-intersecting nodes starting from the leaf nodes. In leaf nodes, we check each pair of primitives for self-intersection. If a node contains self-intersecting primitives, its parent must also include the same self-intersecting primitives. If a node is marked as unsuitable, its parent is also marked as such. For an inner node, the algorithm marks the node as suitable if the node's children do not contain any self-intersecting primitives, and the children nodes do not intersect with each other. To detect the collisions between two children nodes, we use the existing BVH structure. The BVH collision test algorithm (see Procedure BVHCOLLISION in Figure 2) checks whether two BVH nodes are intersecting with each other.¹⁷ A descend rule is used to determine the node that should be descended first.

We use the BVH we constructed in the previous step to find the self-intersection free nodes. A recursive algorithm marks nodes containing no self-intersecting primitives, starting from the leaf nodes. If a node is a leaf node, we test each pair of primitives in the node for self-intersection. For a non-leaf node, it does not contain self-intersecting primitives if both children of the node are non-self-intersecting nodes and the children of the nodes do not intersect with each other.

Third Step: We select the suitable nodes as tetrahedralization leaves. Starting from the root node, a recursive algorithm (see Procedure CONSTRUCTTETRAHEDRA in Figure 2) traverses the BVH and selects nodes that are advantageous to use as tetrahedralization leaves. Using the tetrahedralization nearest-hit cost heuristic, we determine whether a node is beneficial to be tetrahedralized or left as a BVH node (cf. Section 3.3). Given a node, if the approximate average nearest-hit cost of traversing tetrahedralization is less than the SAH cost, we select the node as a tetrahedralization node. If a node is chosen as a tetrahedralization node, we do not check its children. We convert the selected BVH nodes into tetrahedralization leaves by pruning their children and constructing tetrahedralizations for the primitives in the selected nodes. We use the axis-aligned bounding box of BVH nodes to bound each tetrahedralization. We use Tetgen¹⁸ software to tetrahedralize the primitive groups and use the tetrahedron representation in Reference [3] to store tetrahedralizations.

3.2 | Traversal

The nearest-hit traversal for BTH (see Procedure BTHINTERSECT in Figure 2) is similar to the nearest-hit BVH traversal. Starting from the root, we descent the hierarchy until we reach a leaf node. If the bounding box a node does not intersect with the ray, we do not check its children for an intersection. When we encounter a leaf node, we perform different intersection tests based on the leaf node type. If a leaf node does not contain a tetrahedralization, we test for a ray-primitive intersection for all primitives in the node. Otherwise, we traverse the tetrahedralization using the traversal method proposed in Reference [3]. Starting from the initial tetrahedron, we process the tetrahedralization structure, one tetrahedron at a time, until the ray hits a primitive face or exits the tetrahedralization. Nearest-hit traversal on tetrahedralization requires determining the initial tetrahedron that the ray hits. Appendix A provides the details of this process.

3.3 | Nearest-hit cost calculation

BTH construction algorithm builds tetrahedralizations for nodes that are advantageous to be tetrahedralized. Given a BVH node, the algorithm constructs tetrahedralization if the average nearest-hit cost of tetrahedralization of the node's primitives is less than the SAH cost of the corresponding BVH branch. We calculate the approximate cost assuming the rays are distributed uniformly in space and rays originated from the outside of the geometry. The approximate nearest-hit cost on tetrahedralization directly depends on the number of tetrahedra traversed. We define the approximate cost of traversing a tetrahedralization as

$$Cost_{TET}(node) = C_{tet} \times N_{avg}(\mathcal{T}_{node}), \quad (2)$$

where C_{tet} is the cost of traversing a single tetrahedron in tetrahedralization, and $N_{avg}(\mathcal{T}_{node})$ is the average number of tetrahedra traversed during nearest-hit traversal for \mathcal{T}_{node} . To calculate the cost, we first calculate approximation of $N_{avg}(\mathcal{T}_{node})$. Then using Equation 2 we calculate the $Cost_{TET}(node)$. We propose three different ways to approximate $N_{avg}(\mathcal{T}_{node})$;

sampling-based cost calculation, *average depth-based cost calculation*, and *face count-based cost calculation*. Among these, only the sampling-based calculation requires a tetrahedralization to be present. Others, approximate the cost without requiring a tetrahedralization. Appendices B and C describe the nearest-hit cost calculation methods and the nearest-hit cost selection process, respectively.

4 | ANIMATED SCENES

There are two approaches to updating the acceleration structure when the geometry changes in dynamic or animated scenes. The first approach is to rebuild the entire acceleration structure. Although rebuilding provides a simple way to adapt to the changing geometry, it can be an expensive operation for some structures. The second approach is to update the existing acceleration structure by updating parts that changed. Although this is more efficient than rebuilding the entire structure, updates reduce the acceleration structure's quality, reducing the nearest-hit efficiency.

We can categorize dynamic scenes according to the types of motion that the objects perform. In scenes with hierarchical motion, groups of primitives move in the same way. In incoherent motion, the primitives move independently from each other. Depending on the characterization of motion, different algorithms are suitable for adapting acceleration structures.

Because BTH includes a BVH, most of the dynamic BVH algorithms also work for BTH. When modifying a BVH update algorithm into a BTH algorithm, we have to take care of tetrahedralization leaf nodes. One can apply small deformations to a scene without requiring any update to the tetrahedralization.² We will describe how we can adapt BTH to dynamic scenes that perform hierarchical motions.

4.1 | Two-level BTH for hierarchical motion

If the primitives exhibit hierarchical motion, we can use two-level (multi-level) hierarchies¹⁹ as an acceleration structure. For such a two-level hierarchy, we group the primitives and build an acceleration structure for each group. Constructed acceleration structures are the bottom-level of the two-level acceleration structure. When building bottom-level acceleration structures, we use the local reference frames of the objects. Then, we construct a top-level acceleration structure for the bottom-level acceleration structures. In two-level hierarchies, the nearest-hit traversal starts from the node at the top level. When we reach a bottom-level node, we transform the ray into the object's reference frame and test it for an intersection. When a primitive makes a rigid body motion, only the top-level needs to be rebuilt or updated since primitives in the bottom-level acceleration structure do not change. A side benefit of two-level acceleration structures is they support instancing. When the scene contains duplicate objects, we can use the same bottom-level structure to represent the duplicated objects.

We use BVH as the top-level and BTH as the bottom-level acceleration structure for animated scenes (see Figure 1 (b)). For each object in the 3D scene, we construct a BTH structure. Then we combine these bottom-level structures into the top-level using a BVH. For the construction of BVH, we used the midpoint to partition the bottom-level nodes. In each animation frame, we rebuild the top-level BVH using updated coordinates of the objects. The midpoint partitioning scheme allows us to reconstruct the top-level efficiently for each animation frame.

5 | EXPERIMENTAL RESULTS

5.1 | Experimental setup

We conduct experiments on a computer with six cores @3.2 GHz (Intel i7-9750H), 16 GB of main memory. We construct the acceleration structures on a single thread. We render images at 1920×1080 resolution. For ray tracing, we used a multi-threaded 16×16 tile-based rendering method using only primary rays. Our BTH implementation is based on the BVH implementation of Pbrt²⁰ and we use TetGen¹⁸ for tetrahedralizations. We used Aman et al.³ *Tet32* and *Tet20* tetrahedral representations and named our acceleration structures accordingly, that is, *BTH32* and *BTH20*. For comparison, we used BVH and *k*-d tree implementations of Pbrt. We used models from McGuire's Computer Graphics Archive.²¹ The Crown model is from Lubich.²²

5.2 | Results

5.2.1 | Static scenes

We used a variety of scenes to compare the BTH structure to other acceleration structures. Table 1 compares the construction and rendering times of our hybrid acceleration structures (BTH32 and BTH20) against other structures (BVHs and k -d trees) for a set of scenes. The scenes in this table cannot be tetrahedralized directly. Hence, we did not compare tetrahedralization-based acceleration structures in this table. Table 2 compare the construction and rendering times of our hybrid acceleration structures (BTH32 and BTH20) against other acceleration structures, including tetrahedralization-based acceleration structures, BVHs, and k -d trees, for a set of scenes that can be directly tetrahedralized without any preprocessing.

In most cases, BTH performs better than BVH, which is because of the tetrahedralization traversal cost heuristic. As long as the tetrahedralization traversal cost approximation is accurate, the BTH construction algorithm selects the tetrahedralizations that improve the rendering (ray tracing) cost. For some cases, k -d tree performs better than BTH in terms of the rendering cost. In general, k -d tree performs worse than BTH when the scene contains a large amount of intersecting geometry. Many intersecting geometries cause many duplicate primitives in k -d tree structure. Therefore, self-intersections affect k -d tree negatively, for both construction and rendering times. On the other hand, self-intersections have a negligible effect on BTH. If a scene contains a high number of self-intersections, these self-intersecting geometry is stored in BVH leaves in the BTH structure.

Compared to BVH, the BTH construction algorithm is always significantly slower than the BVH construction algorithm because of the need for tetrahedralization. Firstly, TetGen uses algorithms with $\mathcal{O}(N^2)$ worst-case computational complexity. Secondly, robust tetrahedralization with exact arithmetic slows down the tetrahedralization process. In some cases, BTH outperforms the k -d tree. When a scene contains a lot of intersecting geometry, the construction of the k -d tree takes a significant amount of computation time, which we can see in the Hairball model. Since the intersecting geometry is stored in BVH nodes of BTH, BTH performs well.

Rendering using the hybrid BTH structure is faster than rendering with tetrahedralizations. Besides, BTH requires less time to construct in all cases than tetrahedralizations. Another advantage of BTH over tetrahedralizations is that we can build BTH for any 3D geometric models. On the other hand, we can construct tetrahedralizations for non-self-intersecting geometries, or we must perform a self-intersection removal step before tetrahedralization. Experimental results show that BTH20 has a faster rendering speed than BTH32. It is because Tet20 used in BTH20 requires less memory than Tet32 used

TABLE 1 Construction and rendering times (ms) for different acceleration structures on scenes that cannot be directly tetrahedralized. We compare BTH32 and BTH20 with BVH²⁰ and k -d tree²⁰

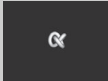



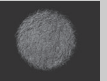








Scenes										
										
No. faces	2,880	345,938	2,505,992	385,162	2,880,000	1,020,907	3,540,310	262,267	9,980,699	1,875,632
Construction times										
BTH32	103.20	496.21	3,558.79	4,293.72	14,527.30	2,873.49	24,365.30	1,944.00	18,569.90	4,690.44
BTH20	101.79	476.33	3,485.51	4,197.93	14,541.70	2,786.16	24,311.90	1,951.73	18,536.50	4,643.17
BVH	2.43	341.56	2,640.36	359.97	2,976.25	1,025.23	3,808.49	245.38	11,185.10	1,546.62
k -d tree	24.26	1,459.58	14,127.60	2,489.23	62,846.80	9,315.58	26,292.00	2,028.37	77,815.50	4,521.61
Rendering times										
BTH32	14.94	80.09	145.74	92.18	274.73	444.60	241.61	378.73	550.58	186.13
BTH20	14.66	76.88	140.46	87.73	266.58	423.91	231.10	349.98	516.49	173.60
BVH	17.50	79.75	145.59	94.36	271.20	459.61	242.66	384.20	586.89	190.15
k -d tree	15.13	95.09	166.78	88.345	313.78	419.74	247.21	213.80	315.46	194.70

TABLE 2 Construction and rendering times (ms) for different acceleration structures on scenes that can be directly tetrahedralized. We compare BTH32 and BTH20 with tetrahedralization-based acceleration structures,³ BVH²⁰ and k -d tree²⁰

Scenes						
						
	Armadillo-2	Mix-2	Mix close			
No. faces	345,938	2,505,992	2,505,992			
	Construction times			Rendering times		
	Armadillo-2	Mix-2	Mix Close	Armadillo-2	Mix-2	Mix Close
BTH32	2,398.8	4,473.5	4,593.5	79.54	142.37	232.23
BTH20	3,367.7	4,485.8	4,468.8	77.95	140.70	218.13
Tet32	8,392.1	89,949.7	91,551.4	150.84	312.55	363.67
Tet20	8,380.4	90,291.4	90,564.1	118.28	252.76	273.62
BVH	373.1	2,683.6	2,683.3	78.71	143.45	224.06
k -d tree	1,460.6	14,830.4	14,090.6	98.18	164.56	254.79

in BTH32. The small memory requirement of the tetrahedron representation of BTH20 leads to higher cache utilization than that of BTH32. BTH32 and BTH20 structures require approximately similar construction times.

5.2.2 | Dynamic scenes

Figure 3 show still frames from the animations of the rotating armadillo, the random motion, and the falling hairballs, respectively, rendered using the two-level BVH-BTH structure. Please see the supplementary video for these animations.

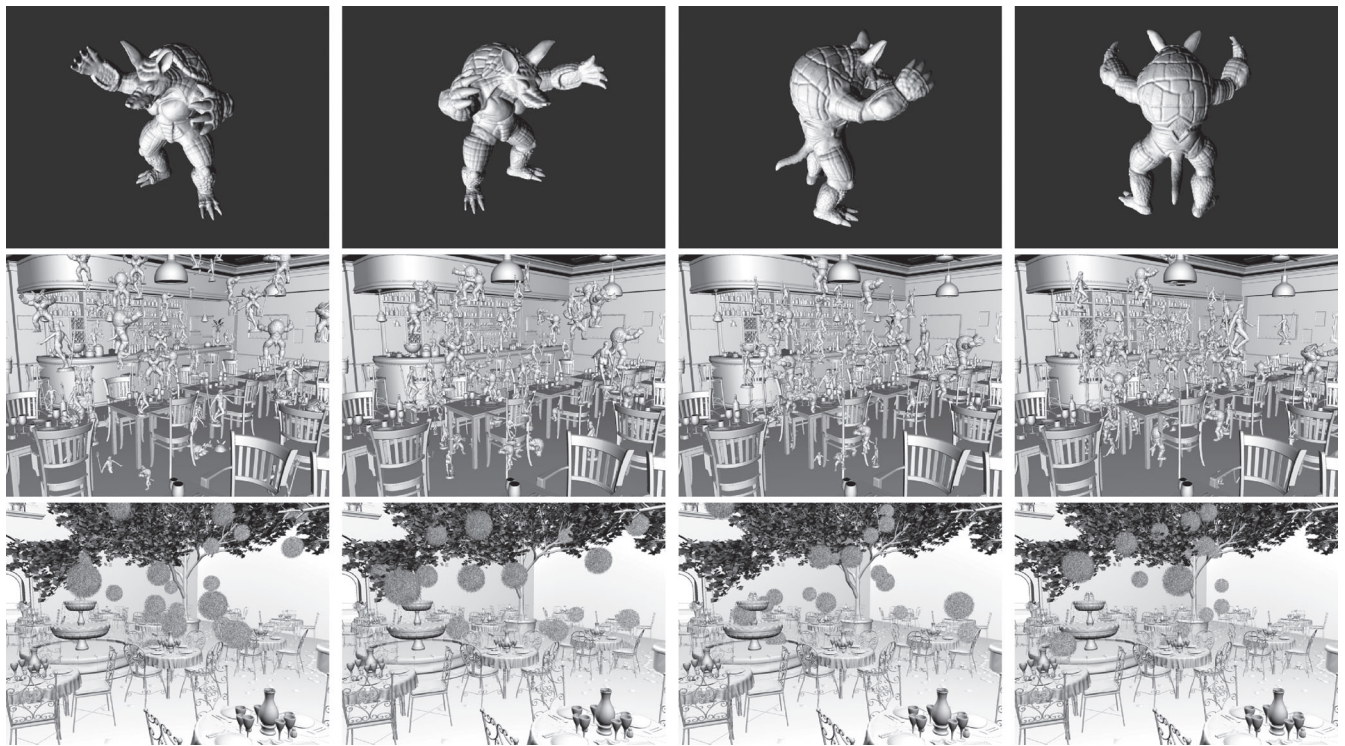


FIGURE 3 Still frames from the animated scenes rendered using the two-level BVH-BTH structure. First row: Rotating Armadillo. Second row: Random Motion. Third row: Falling hairballs.

TABLE 3 Construction and average rendering times per frame (ms) of different acceleration structures for the animated scenes. We compare our BTH32 and BTH20 structures with BVH.²⁰ See Figure 3 for the models.

	Scenes					
	Rotating Armadillo		Random Motion		Falling Hairballs	
No. faces	345,938		1,020,907		12,860,699	
No. objects	1		101		101	
	Construction times			Average rendering times (per frame)		
	Rotating Armadillo	Random Motion	Falling Hairballs	Rotating Armadillo	Random Motion	Falling Hairballs
BTH32	2,403.3	8,831.5	37,920.2	70.88	675.51	721.06
BTH20	2,466.7	9,075.9	38,030.4	70.14	659.55	699.02
BVH	363.14	1,609.1	14,409.5	70.84	671.56	737.32

We compared the performance of the two-level BVH structure against that of the two-level BVH-BTH structure. Table 3 shows the results. In general, the proposed two-level BVH-BTH structure using BTH20 at the lower level is faster than the two-level BVH structure.

6 | CONCLUSIONS AND FUTURE RESEARCH

We propose the BVH-Tetrahedralization Hybrid structure for accelerating the nearest-hit tests for the ray tracing algorithm. We tested our acceleration structure with different scenes. We showed that we could improve the BVH structure's rendering time by converting it into a BTH hybrid structure. Our experiments show that, in all cases, the proposed BTH20 acceleration structure outperforms the BVH structure in terms of rendering times at the cost of slower construction times. We proposed two methods for approximating average nearest-hit costs for tetrahedralization structures. We show that the proposed cost calculation methods can provide good approximations for the average nearest-hit cost on tetrahedralized scenes.

Additionally, we can use the two-level acceleration structure for dynamic scenes with hierarchical motions. Our experiments show that the two-level BVH-BTH outperforms the two-level BVH-BVH for the tested scenes where the objects perform hierarchical movements. Some possible future work areas are as follows:

1. *Adapting BTH for scenes with deforming geometries:* One advantage of using tetrahedralizations in BTH is that they do not require refitting for small deformations. For animated scenes with deforming geometry, we could use BVH update methods to update BTH. We could also use BTH for rendering animated frames of articulated bodies with nonrigid limbs.
2. *Exploiting ray connectivity for secondary rays:* In hierarchical acceleration structures, it is hard to exploit ray connectivity. The nearest-hit test starts from the root of the hierarchy for secondary rays. On the other hand, tracing secondary rays is easy on tetrahedralizations. After a primary ray hits a surface, secondary rays can continue from there without an initialization step. Similarly, we can trace secondary rays in tetrahedralizations on the leaves of a BTH hierarchy.

ACKNOWLEDGMENTS

This research is supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under No. 117E881.

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Aytek Aman  <https://orcid.org/0000-0002-4712-2561>

Serkan Demirci  <https://orcid.org/0000-0001-8805-5310>

Uğur Gündükbay  <https://orcid.org/0000-0003-2462-6959>

Ingo Wald  <https://orcid.org/0000-0003-0046-713X>

REFERENCES

1. Kajiya JT. The rendering equation. *ACM Comp Graph (Proc SIGGRAPH '86)*. 1986;20(4):143–50.
2. Lagae A, Dutré P. Accelerating ray tracing using constrained tetrahedralizations. *Comp Graph Forum*. 2008;27(4):1303–12.
3. Aman A, Demirci S, Gündükbay U. Compact tetrahedralization-based acceleration structure for ray tracing. 2021. arxiv preprint:arXiv:2103.02309.
4. Fujimoto A, Tanaka T, Iwata K. ARTS: accelerated ray-tracing system. In: Joy KI, Grant CW, Max NL, Hatfield L, editors. *Tutorial: computer graphics; image synthesis*. New York, NY: Computer Science Press; 1988. p. 148–59.
5. Cleary JG, Wyvill G. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Vis Comp*. 1988;4(2):65–83.
6. Bentley JL. Multidimensional binary search trees used for associative searching. *Commun ACM*. 1975;18(9):509–17.
7. Goldsmith J, Salmon J. Automatic creation of object hierarchies for ray tracing. *IEEE Comp Graph Appl*. 1987;7(5):14–20.
8. MacDonald DJ, Booth KS. Heuristics for ray tracing using space subdivision. *Vis Comp*. 1990;6(3):153–66.
9. Wald I, Havran V. On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. Salt Lake City, UT; 2006. p. 61–9.
10. Müller G, Fellner DW. Hybrid scene structuring with application to ray tracing. *Proceedings of the International Conference on Computer Vision*. Goa, India; 1998. p. 19–26.
11. Wald I, Boulos S, Shirley P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans Graph*. New York, NY: Association for Computing Machinery; 2007;26(1):6–18.
12. Popov S, Gunther J, Seidel HP, Slusallek P. Experiences with streaming construction of SAH KD-trees. *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. Salt Lake City, UT: IEEE; 2006. p. 89–94.
13. Wald I. On fast construction of SAH-based bounding volume hierarchies. *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. RT '07. Ulm, Germany: IEEE Computer Society; 2007. p. 33–40.
14. Miller GL, Talmor D, Teng SH, Walkington NJ, Wang H. Control volume meshes using sphere packing: generation, refinement and coarsening. *Proceedings of the 5th International Meshing Roundtable*. Pittsburgh, PA; 1996. p. 47–61.
15. Hu Y, Zhou Q, Gao X, Jacobson A, Zorin D, Panozzo D. Tetrahedral meshing in the wild. *ACM Trans Graph*. 2018;37(4):60 14 pages.
16. Maria M, Horna S, Aveneau L. Efficient ray traversal of constrained delaunay tetrahedralization. *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*. Porto, Portugal; vol. 1 of VISIGRAPP '17; 2017. p. 236–43.
17. Ericson C. *Real-time collision detection*. San Francisco, CA: CRC Press; 2004.
18. Si H. TetGen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans Math Softw*. 2015;41(2):11 36 pages.
19. Lext J, Akenine-Möller T. Towards rapid reconstruction for animated ray tracing. *Proceedings of the Eurographics 2001–Short Presentations*; Montreal, Canada: Eurographics; 2001. p. 311–8.
20. Pharr M, Jakob W, Humphreys G. *Physically based rendering: from theory to implementation*. 3rd ed. San Francisco, CA: Morgan Kaufmann; 2016.
21. McGuire M. *Computer graphics archive*. 2017. <https://casual-effects.com/data>. Accessed 03 Mar 2021.
22. Lubich M. *Blender minutes*. 2012. <https://www.loramel.net/>. Accessed 03 Mar 2021.

AUTHOR BIOGRAPHIES



Aytok Aman received the B.S. and M.S. degrees in computer engineering from Bilkent University, Ankara, Turkey, in 2011 and 2014, respectively. He is currently working toward a Ph.D. degree in computer engineering at Bilkent University. His research interests include computer graphics, specifically rendering techniques for three-dimensional scenes, virtual and augmented reality, crowd simulation, computer vision, and computational geometry.



Serkan Demirci received the B.S. and M.S. degrees in computer engineering from Bilkent University, Ankara, Turkey, in 2017 and 2020, respectively. He is currently working toward a Ph.D. degree in computer engineering at Bilkent University. His research interests include computer graphics, specifically rendering techniques for three-dimensional scenes, virtual and augmented reality, crowd simulation, and computational geometry.



Uğur Güdükbay received the B.S. degree in computer engineering from the Middle East Technical University, Ankara, Turkey, in 1987, and the M.S. and Ph.D. degrees in computer engineering and information science from Bilkent University, Ankara, Turkey, in 1989 and 1994, respectively. He conducted research as a Postdoctoral Fellow at the Human Modeling and Simulation Laboratory, the University of Pennsylvania. Currently, he is a professor in the Department of Computer Engineering, Bilkent University. His research interests are different aspects of computer graphics, including human modeling and animation, crowd simulation, physically-based modeling, rendering, and visualization.



Ingo Wald is a director of ray tracing at NVIDIA. He received his master's degree from Kaiserslautern University and his Ph.D. from Saarland University, then served as a post-doctorate at the Max-Planck Institute Saarbrücken, as a research professor at the University of Utah, and as technical lead for Intel's software-defined rendering activities (in particular, Embree and OSPRay). Ingo has co-authored more than 90 papers, multiple patents, and several widely used software projects around ray tracing.

SUPPORTING INFORMATION

Additional supporting information may be found online in the Supporting Information section at the end of this article.

How to cite this article: Aman A, Demirci S, Güdükbay U, Wald I. Multi-level tetrahedralization-based accelerator for ray-tracing animated scenes. *Comput Anim Virtual Worlds*. 2021;32:e2024. <https://doi.org/10.1002/cav.2024>