Ray Tracing Massive Amounts of Animated Geometry

HOLGER GRUEN, Advanced Micro Devices, Inc., Germany
CARSTEN BENTHIN, Advanced Micro Devices, Inc., Germany
MICHAEL KERN, Advanced Micro Devices, Inc., Germany
DAVID MCALLISTER, Advanced Micro Devices, Inc., USA

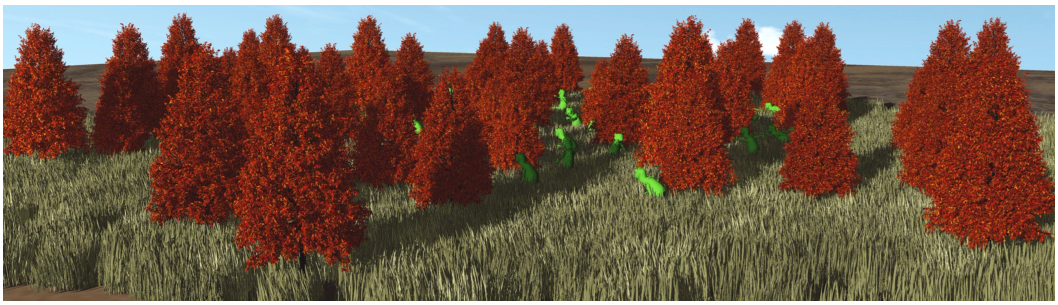

Fig. 1. Our tetrahedral cage-based method allows for ray tracing 585 million animated triangles (uniquely animated objects, no reuse through instancing) at 60 frames per second on an AMD RX 9070 XT at 1080p.

We present a method for efficiently ray tracing large amounts of uniquely animated complex mesh-based geometry. A tetrahedral cage is used to split the original non-animated triangle geometry into disjoint sets of triangles. For each tetrahedron, an acceleration structure is built over its enclosed triangles, and a separate one over all tetrahedra. Instead of applying animation transformations to the actual triangular geometry, we apply them only to the tetrahedral cage and transform rays intersecting the animated tetrahedra. Only the acceleration structure over all tetrahedra needs to be rebuilt once per frame. The cost of animating geometry per frame depends solely on the resolution of the tetrahedral cage and is decoupled from the density of the original geometry. Our method targets animations that preserve triangle connectivity and approximates vertex motion by a piecewise-linear deformation induced by the tetrahedral cage. We demonstrate that our method outperforms competing animation approaches in both update time and memory consumption, enabling ray tracing of large amounts of animated complex geometry in real time on current generation GPUs.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; • **Theory of computation** → **Sorting and searching**; **Massively parallel algorithms**.

Additional Key Words and Phrases: animation, bounding volume hierarchy, ray tracing

ACM Reference Format:
Holger Gruen, Carsten Benthin, Michael Kern, and David McAllister. 2026. Ray Tracing Massive Amounts of Animated Geometry. *Proc. ACM Comput. Graph. Interact. Tech.* 9, 4, Article 49 (July 2026), 18 pages. https://doi.org/10.1145/3820014

Authors' Contact Information: Holger Gruen, Advanced Micro Devices, Inc., Germany, holger.gruen@amd.com; Carsten Benthin, Advanced Micro Devices, Inc., Germany, carsten.benthin@amd.com; Michael Kern, Advanced Micro Devices, Inc., Germany, michael.kern@amd.com; David McAllister, Advanced Micro Devices, Inc., USA, david.mcallister@amd.com.

1 Introduction

Real-time ray tracing of animated geometry, in particular animated triangle meshes, requires not only animating the actual triangles per frame but also updating or rebuilding the acceleration structure over the triangles. This process has high computational and memory costs. The costs scale directly with the amount of animated triangles and are therefore limited per frame to avoid exceeding the given frame time and memory budget. Given the constraint of maintaining real-time frame rates of 60 frames per second or more, only a few milliseconds can be afforded to animate geometry and to update acceleration structures each frame. Techniques like using lower resolution geometry level of detail (LOD), partial (staggered) acceleration structure updates, or selective reuse of animated geometry instead of applying unique animations are common mitigation strategies to lower the per-frame costs.

We propose a method to address the issue of a limited budget of animated geometry per frame by decoupling the cost of animating geometry from the triangle density. We focus on *deformable* animation, where the triangle connectivity remains unchanged, while the vertex animation is approximated by a piecewise-linear deformation induced by a tetrahedral cage. Our method builds a low-resolution tetrahedral cage over the original non-animated geometry in a preprocessing step, splitting it into small disjoint sets of triangles. An acceleration structure is built over all triangles within a tetrahedron and a top-level acceleration structure is built over all tetrahedra. This creates a two-level acceleration structure hierarchy. Instead of applying animation transformations to the triangles per frame, we apply them only to the tetrahedral cage and update only the top-level acceleration structure per frame. The set of triangles within a tetrahedron and its acceleration structure are never modified. A ray is traced through the top-level acceleration structure first, and for every intersected tetrahedron, a uniquely deformed copy of the ray, based on the transformed tetrahedron, is created and used to intersect the static bottom-level acceleration structure within.

The resolution of the tetrahedral cage is independent of the triangle density of the animated object, and the number of tetrahedral cage triangles is significantly smaller than the total amount of triangles in the object. This reduces the per-frame animation cost to only a fraction of the cost of traditional approaches, which modify all object triangles and update the corresponding acceleration structure. At the same time, our approach requires significantly less memory. Savings are particularly high when copies of the same triangular object are animated multiple times but with different animations. Given the memory and time savings, our approach is able to ray trace massive amounts of uniquely animated geometries in real time which has not been possible before.

Our contributions include:

- Applying tetrahedral cages to decouple animation complexity, BVH memory footprint, and update time from triangle density when ray tracing animated geometry.
- Two new algorithms for ray tracing animated geometry using tetrahedral cages: the first exploits ray-instance transformations for efficiency, the second guarantees watertightness using relative 4D geometry and bounding volume encodings.
- A mapping of our algorithms to the DXR pipeline, allowing easy adoption on current generation GPUs.

2 Related Work

Ray tracing animated content requires updating the underlying acceleration structure. In recent years, the bounding volume hierarchy (BVH) has been established as the dominant acceleration structure for ray tracing, and researchers have proposed many different BVH build algorithms [Meister et al. 2021]. For animations with limited motion, a BVH update, which refits the BVH from the leaves to the root, has been the preferred choice due to its low cost. In order to support arbitrary

geometry transformations, e.g., an object exploding into pieces, a more costly BVH rebuild is required to maintain acceleration structure quality. Alternative approaches try to reduce BVH update/rebuild cost by employing lazy build strategies [Hunt et al. 2007; Lee and Liktor 2020] or by incrementally modifying the existing BVH's topology [Karras and Aila 2013; Kensler 2008].

Real-time ray tracing APIs like DXR [Microsoft 2020] or Vulkan-RT [Khronos Group 2020] use a two-level acceleration structure hierarchy: a bottom-level acceleration structure (BLAS) per object and a top-level acceleration structure (TLAS) over instances of objects. A BLAS rebuild is required when triangles within an object change, and a TLAS rebuild is required when instances of objects change, or when the object itself changes. Most DXR or Vulkan implementations use BVHs as acceleration structures. Depending on API quality flags, the implementation can decide whether to perform a full BVH rebuild or a simpler BVH update.

Displaced micro-meshes (DMMs) ([Maggiordomo et al. 2023]) avoid the construction of the BVH over the geometry by providing an implicitly defined BVH over the hierarchically encoded micro-mesh. Gruen et. al. [2024] proposed a method to improve the quality of DMMs in the context of animation. DMMs have not been widely adopted, as their representation is tied to piecewise manifold objects, which limits their applicability when dealing with real-world content.

Building triangle meshes out of small triangle clusters instead of individual triangles has become increasingly common in recent years due to cluster-based geometry LOD rendering [Brian Karis 2022]. Ray tracing of cluster-based LODs has been shown [Benthin and Peters 2023] and real-time API support has been proposed [Microsoft 2026; NVIDIA Corporation 2024]. Clusters introduce another level into the existing two-level hierarchies and therefore offer a finer granularity when only parts of objects are affected by animation.

For scenes with a high instance/object count, rebuilding the top-level BVH per frame can be a limiting factor. Top-level partitions [Khronos Group 2024] help reduce BVH build complexity by classifying sets of objects as affected by changes in the scene or not. This prevents rebuilding or updating the entire BVH when only a subset of instances/objects have been modified.

Tetrahedral cages, as a spatial acceleration structure for 3D objects, have been widely used in the context of animation, deformation, simulation [Alliez et al. 2005; Jacobson et al. 2011; Joshi et al. 2007; Mezger et al. 2003; Morrical et al. 2023; Müller et al. 2005] and in ray-tracing and acceleration-structure contexts, including approaches that employ non-orthogonal local parameterizations for traversal [Aman et al. 2022; Kácerik and Bittner 2025; Lagae and Dutré 2008]. Tetrahedra offer a balance between geometric simplicity and the directional tightness of an oriented bounding box. Very recently, Luton et. al. [Luton and Tricard 2025] presented a method to rasterize animated but meshless object representations using tetrahedral cages. Our method follows their general idea of applying the deformation transformation to the non-animated pose of the geometry but extends it to the real-time ray tracing domain.

The cost of updating or rebuilding BVH structures is directly related to the actual amount of animated geometry. When animating a large number of complex objects with unique deformations, this can quickly become the limiting factor in achieving real-time performance. Even worse, each uniquely animated object also needs a copy of the original geometry and its acceleration structure, which can lead to excessive memory consumption.

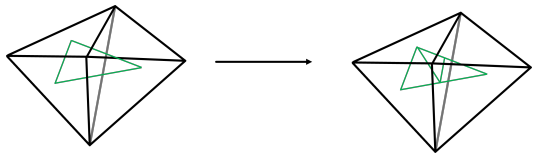

Fig. 3. A triangle overlapping two tetrahedra is clipped, and the resulting polygons are triangulated.

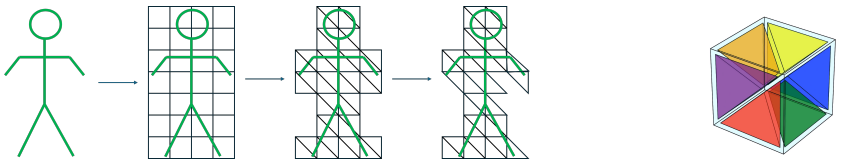

Fig. 2. 2D illustration of our tetrahedralization algorithm (left to right): the object in the non-animated pose, voxelization using a regular grid, removal of empty voxels and splitting of non-empty voxels into tetrahedra; tetrahedra that do not intersect any geometry are removed. (right-most) Each voxel is split into six tetrahedra.

3 Building Tetrahedra Cages

Our approach uses a tetrahedral cage to split a triangular object into smaller disjoint sets of triangles. Various algorithms exist [Si 2006; Wang et al. 2020; Wang and Yu 2012; Yu et al. 2024] to build such a tetrahedral cage. We adopt a simple voxel-based approach that runs in multiple stages (see Figure 2). We start with the object in its non-animated pose and use a regular grid to subdivide it into voxels. All empty voxels are removed and the remaining voxels are split into tetrahedra. In the final step, all tetrahedra that do not intersect any object geometry are removed. In our implementation, each voxel is split into six tetrahedra [Domiter and Žalik 2008; Freudenthal 1942].

Given the set of non-empty tetrahedra, we determine the set of triangles intersecting each tetrahedron and clip these against the tetrahedron's four bounding triangles (see Figure 3). This clipping process creates a small triangle mesh, a *μMesh*, per tetrahedron. Clipping a triangle that overlaps adjacent tetrahedra is required to avoid artifacts, as each tetrahedron will deform the same triangle differently. Our implementation makes sure that vertices introduced through clipping are deduplicated and are shared between the current and all adjacent tetrahedra. The clipping process is necessary to ensure the following bounding property: all vertices of the *μMesh* are contained within the volume of the corresponding tetrahedron. Vertices shared with neighboring tetrahedra are located exactly on the bounding triangles of the tetrahedron. Please note that clipping introduces new triangles and vertices, therefore increasing their total counts (see Section 7).

Once all *μMeshes* are created, a μBLAS is created per *μMesh*. A μBLAS is a bottom-level acceleration structure built over the triangles in a *μMesh*. Each μBLAS is created once and will remain static (never modified in any way). Note that additional optimizations for static μBLAS such as compaction could be applied at this point. The same tetrahedral cage and therefore all associated *μMeshes* and μBLAS objects can be reused by uniquely animated instances of the same object (see Section 7). In a final step, a top-level acceleration structure is built over all tetrahedra. We will call this a *tet*LAS, and call the mesh consisting of the bounding triangles of all tetrahedra a *tet*Mesh. Note that a *tet*LAS has tetrahedra at the leaf level and each tetrahedron refers to a μBLAS object (see Figure 4).

Our current implementation does not enforce a fixed triangle budget per tetrahedron. The number of triangles per μBLAS is determined by the cage resolution and geometry distribution.

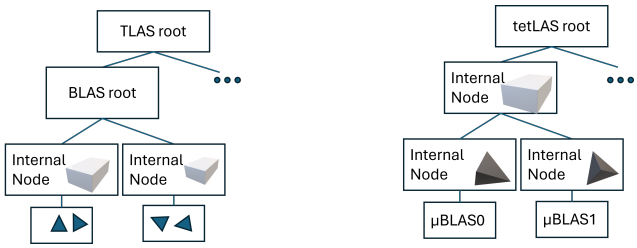

Fig. 4. (left) Regular BVH-based acceleration structure where internal nodes either refer to other internal nodes or leaves containing triangles. (right) In a *tet*LAS, instead of referencing triangles, internal nodes can refer to tetrahedra and those to μBLASes.

4 Animating and Intersecting Tetrahedra Cages

A tetrahedral cage (*tet*Mesh) that encapsulates a set of triangles offers an efficient way to approximate any kind of deformable animation (with triangle connectivity unchanged) applied to the vertices of the triangles themselves. Instead of modifying the triangle vertices, our method only modifies the vertices of the *tet*Mesh, which results in a piecewise-linear spatial deformation. For our test scenes (see Section 7), we either animate the *tet*Mesh vertices by directly applying a transformation function to each vertex or by using a bone-based animation scheme. After modifying the *tet*Mesh vertices, the *tet*LAS is updated. As the number of tetrahedra is orders of magnitude smaller than the total number of triangles (see Section 7), the cost of updating the *tet*LAS is significantly smaller than updating a regular BLAS over all triangles. Modifying the *tet*Mesh vertices and updating the *tet*LAS has to be done once per frame before rendering.

During rendering, a ray in world-space is traced through the *tet*LAS first. Once a tetrahedron is hit, the ray is transformed from world-space into the tetrahedron's local space (and the associated μBLAS). Following that, the transformed ray is traversed through the μBLAS. Our method transforms the ray from world-space into the tetrahedron's local space by expressing the transformation as a 3D basis-change operation (see Figure 5 and Section 4.1). Even though this operation is very efficient, it cannot guarantee watertightness in all cases. We therefore propose another variant (see Section 5) that keeps the ray in world-space and instead transforms geometry and acceleration structure data from a 4D representation into world-space during ray traversal.

In general, the orders of magnitude lower complexity of *tet*Mesh and *tet*LAS in comparison to the original triangular object allows us to significantly reduce memory consumption, animation complexity, and acceleration structure build times (see Section 7). In particular, the μBLAS objects can be reused for different deformations of the same object. Each unique deformation only requires a copy of the *tet*Mesh, which we call *tet*AnimCopy.

4.1 Intersecting Tetrahedra using Ray-Instance Transformations

Modern ray tracing APIs use a two-level TLAS/BLAS hierarchy in which the TLAS is built over instanced BLAS objects. Instances can be transformed by a 3×4 matrix that transforms a ray from world-space into the local space of the BLAS object. Note that real-time ray tracing APIs [Khronos Group 2020; Microsoft 2020] require specifying the inverse matrix. In the following, we will describe how to exploit the instance transformation to transform a ray from world-space into the local tetrahedron space (see Figure 5).

The four vertices, v_0, v_1, v_2 and v_3 (see right-hand side of Figure 5), of each non-deformed tetrahedron, define three 3D basis vectors of a non-orthogonal coordinate system. If we subtract v_0 from every vertex of a *μMesh*, then all resulting vertices can be written as a linear combination of $v_1 - v_0$,

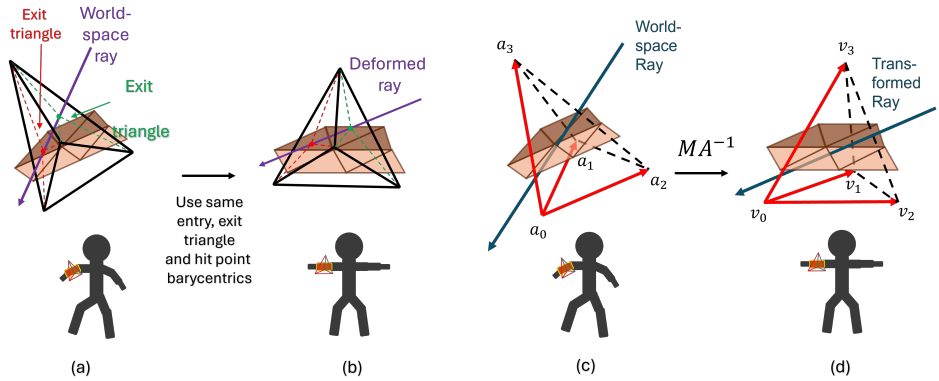

Fig. 5. (a) An animated tetrahedron that is hit by a ray in world-space. We determine the entry and exit triangles of the hit animated tetrahedron and the two barycentric coordinate sets of the entry and exit points. (b) These barycentric coordinates are used to compute entry and exit points on the same triangles of the non-animated tetrahedron. These two new points define our new transformed ray. (c) The four vertices of a (non-degenerate) animated tetrahedron form a basis of a non-orthogonal coordinate system (see Equations 1 and 2). We can express steps (a) and (b) of transforming the ray from world to local space as a basis-change operation (d).

$v_2 - v_0$, and $v_3 - v_0$. Those can be used to form a 3×3 matrix, as shown in Equation 1. We assume that these vectors are linearly independent for now.

$$M = \begin{pmatrix} v_{1_x} - v_{0_x} & v_{2_x} - v_{0_x} & v_{3_x} - v_{0_x} \\ v_{1_y} - v_{0_y} & v_{2_y} - v_{0_y} & v_{3_y} - v_{0_y} \\ v_{1_z} - v_{0_z} & v_{2_z} - v_{0_z} & v_{3_z} - v_{0_z} \end{pmatrix} \tag{1}$$

In a similar vein, we can define a 3×4 matrix A from the four world-space vertices (a_0, a_1, a_2 and a_3) of the animated version of the same tetrahedron, as shown in Equation 2. The first three columns of A define the basis of the deformed coordinate system. The fourth column moves the origin of the animated coordinate system to a_0.

$$A = \begin{pmatrix} a_{1_x} - a_{0_x} & a_{2_x} - a_{0_x} & a_{3_x} - a_{0_x} & a_{0_x} \\ a_{1_y} - a_{0_y} & a_{2_y} - a_{0_y} & a_{3_y} - a_{0_y} & a_{0_y} \\ a_{1_z} - a_{0_z} & a_{2_z} - a_{0_z} & a_{3_z} - a_{0_z} & a_{0_z} \end{pmatrix} \tag{2}$$

Transforming the non-animated vertices in the static μBLAS with respect to the four vertices of the animated tetrahedron becomes a simple basis-change operation. Therefore, we choose $A(M^{-1})_{3x4}$ as the instance transform for the μBLAS. Here, $(M^{-1})_{3x4}$ is constructed by inverting M and extending M^{-1} to a 3×4 matrix by adding a fourth column of zeros. A requirement of the instance transformation-based approach is that all non-animated tetrahedra need to be non-degenerate. As the tetrahedra are generated in a preprocessing step, we assume that the generation of degenerate tetrahedra can be prevented. Another restriction is that all μBLAS instance transforms must be invertible, i.e., $(A(M^{-1})_{3x4})^{-1}$ or MA^{-1} must exist.

The instance transformation-based approach cannot guarantee watertightness as an animated object will consist of instances with varying transforms. Each instance will use a different ray when intersecting the local μBLAS. We mitigated this problem by slightly enlarging all tetrahedra before

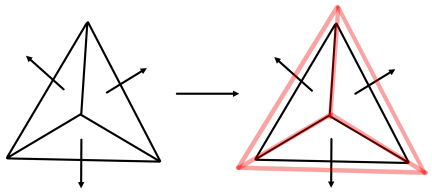

Fig. 6. For the instance transformation-based approach, possible watertightness issues are reduced by slightly growing a tetrahedron along its unit face normals before clipping triangles against.

we clip the original triangles. Each tetrahedron is enlarged by moving each of the four planes that define a tetrahedron outward along its outward-pointing unit-normal by a small epsilon, as shown in Figure 6. This creates overlapping surface geometry where the clipped triangles share an edge. The epsilon is user-defined and scene-dependent, as it must be just large enough so that the overlapping geometry does not create any visible rendering artifacts or too many duplicated tetrahedra intersections (see Section 8), while at the same time avoiding most of the watertightness issues. For our test scenes (see Section 7), we use an epsilon of 2.5×10^{-6}. A solution that guarantees watertightness is presented in Section 5. We tested our mitigation approach in isolation for a sphere model (radius 100, ~1M original triangles, ~1.2M triangles after clipping, $9 \times 12 \times 9$ voxel cage), tracing rays from the sphere origin outwards. Without our mitigation $\sim$ 0.01% of all rays pass through the closed sphere, while with our mitigation, no watertightness escapes occur.

5 Alternative Approach Guaranteeing Watertightness

Watertightness issues for our tetrahedral cage-based traversal, where for example, a ray shoots through a shared edge between adjacent triangles, are caused by different tetrahedra applying different world-to-local-space transformations to the world-space ray, creating different rays in local space, which are then traversed through the local μBLAS. These different local rays can provide inconsistent ray-triangle intersection results for vertices, edges, and triangles shared by different tetrahedra. The clipping process (see Section 3) will make sure that no single triangle overlaps multiple tetrahedra, and that inconsistency can only occur at the shared bounding triangles of two adjacent tetrahedra, e.g., for a shared triangle edge inside a tetrahedron's bounding triangle. However, numerical inaccuracies can be introduced by the clipping process itself, which could be another source of watertightness problems. These could occur when a triangle of the original mesh that passes through the shared bounding triangle of two adjacent tetrahedra is clipped, creating a triangle edge that lies exactly on the shared bounding triangle. The limited accuracy of floating-point representation does not always allow us to accurately represent vertex positions that are exactly on a shared bounding triangle between two adjacent tetrahedra (see Figure 7). The new vertices created in the clipping process might be slightly below or above the shared bounding triangle. This breaks the tetrahedron bounding property because parts of the clipped triangle are located outside the tetrahedron.

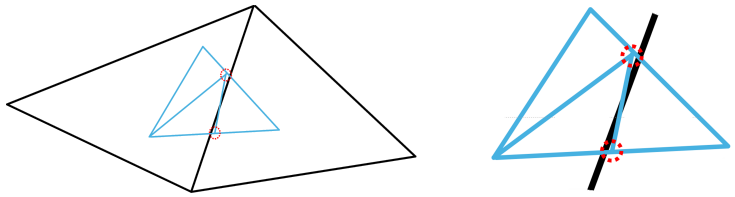

Fig. 7. 2D illustration: (left) The blue triangle is clipped with respect to the shared bounding triangle of two adjacent tetrahedra, creating two new vertices. These new vertices will not always be exactly on the shared bounding triangle but may be slightly off. (right) Zoom: the upper clipped vertex representation lies within the left tetrahedron by a small floating-point margin, while the lower clipped vertex lies within the right tetrahedron. This breaks the bounding property of the tetrahedron, as parts of the clipped triangle are outside. A ray that only intersects the left tetrahedron close to the shared bounding triangle could miss the clipped triangle at the far right.

To guarantee watertightness, we need to perform multiple steps: (a) restore the bounding property of tetrahedra with respect to the contained triangles, (b) provide a consistent view of clipped vertices, edges, and triangles located at the shared bounding triangle between adjacent tetrahedra, and (c) perform consistent ray-triangle intersection tests using the world-space ray and triangle vertices. We will present a method that achieves these three requirements next.

5.1 4D Barycentric Coordinates

3D barycentric coordinates within a triangle extend to 4D barycentric coordinates for a tetrahedron. Any 3D point p within the tetrahedron can be represented as barycentric linear combinations of the four vertices (v_0, v_1, v_2 and v_3) defining the tetrahedron, as shown in Equation 3.

$$p = b_0 \cdot v_0 + b_1 \cdot v_1 + b_2 \cdot v_2 + b_3 \cdot v_3 \tag{3}$$

This allows us to express all 3D points inside the tetrahedron (and on the bounding planes) using four barycentric coordinates b_0, b_1, b_2 and b_3. The key idea to achieve watertightness is to express all triangle vertices as 4D barycentric coordinates relative to the four tetrahedron vertices.

We need to make sure that all points shared by two adjacent tetrahedra, i.e., all points inside a shared bounding triangle, can be reconstructed bitwise identical, regardless of which of the two tetrahedra is used. This requires a unique global ordering of the tetrahedra vertices. After clipping the original mesh triangles to each tetrahedron, we deduplicate vertices from all resulting *μMeshes* and assign a unique ID to each vertex. For each tetrahedron, we sort the four vertices based on their unique IDs in ascending order and use this order each time we evaluate Equation 3.

When computing the 4D barycentric coordinates for each vertex of a clipped *μMesh*, we have to deal with the following cases for these vertices:

(1) The *μMesh* vertex matches one of the tetrahedron's vertices: only the barycentric coordinate that corresponds to the vertex is set to 1. The other three coordinates are set to zero.
(2) The *μMesh* vertex lies exactly on one of the edges of the tetrahedron: only the barycentric coordinates that correspond to the edge vertices are set to non-zero values. The other two coordinates are set to zero. The sum of the two non-zero coordinates must be one.
(3) The *μMesh* vertex lies exactly inside one of the bounding triangles of the tetrahedron. The three barycentric coordinates that correspond to the bounding triangle vertices are set to non-zero values. The remaining coordinate is set to zero. Note that the sum of the three non-zero barycentric coordinates must be one.

(4) The *μMesh* vertex lies fully inside one of the tetrahedra of the cage: all four barycentric coordinates are non-zero and must sum to one.

We use a small, user-defined distance threshold to detect if a vertex is very close to a *tet*Mesh vertex, edge, or triangle. If the vertex is close enough, we snap it to the closest vertex, edge, or triangle and use the snapped position to compute barycentric coordinates.

Given barycentric coordinates and ordered evaluation of Equation 3, we can reconstruct the bit-exact world-space positions of any point within shared bounding triangles, regardless of which of the adjacent tetrahedra we use for the evaluation. We use the relative 4D barycentric encoding for all position data of the non-animated *μMesh* and the corresponding *μ*BLAS, i.e., triangle vertices and bounding volumes. As the triangle vertices are expressed as relative 4D barycentric coordinates, the acceleration structure has to bound relative 4D positions as well. Therefore, we use a binary bounding volume hierarchy with 4D axis-aligned bounding volumes. We call this a 4D BVH.

When a ray intersects an animated tetrahedron (see Section 6), we traverse the associated 4D BVH. At each traversal step, we on-the-fly reconstruct 3D world-space AABBs from the relative 4D representations (applying the *min-sum heuristic* [Nießner 2013] for tighter 3D bounds), given the modified vertices of the tetrahedron. A similar step is performed for the triangle vertices at the 4D BVH leaf level. The relative 4D BVH representation ensures tetrahedron-transformation independence. Appendix A presents a proof that regardless of how the tetrahedron vertices change, the conversion from a relative 4D BVH to a world-space 3D BVH preserves the bounding property. The conversion from relative 4D to 3D world-space is required to guarantee watertightness for ray-primitive intersection tests, as these need to be performed in the same space as the world-space ray.

6 DXR Implementation

The standard approach to uniquely animate a triangular object is to first transform its triangle vertices and then update the corresponding BLAS. Once this process is done for all animated objects and BLASes in the scene, the TLAS over all object/BLAS instances has to be updated as well. The resulting hierarchy structure for the standard animation approach, with a single instance per animated object, is shown in Figure 8.

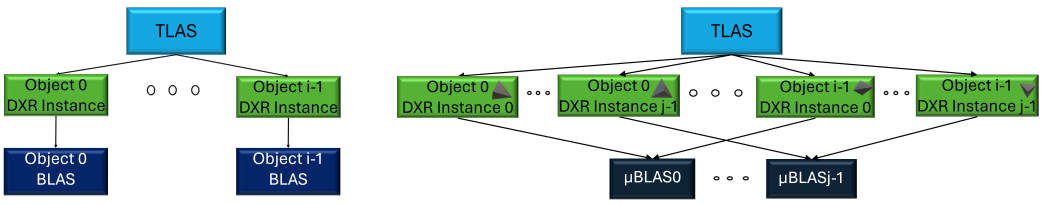

Fig. 8. (left) TLAS/BLAS hierarchy of the standard animation approach, here with a single instance per uniquely animated object. The TLAS is built over all BLAS instances. Animation transformation is first applied to all object vertices, then the BLAS is updated. Once all BLASes have been updated, the TLAS is updated. (right) Different animations of the same object have their own *tet*AnimCopy (a copy of the tetrahedral cage) but share the same *μ*BLAS object. Each *tet*AnimCopy is mapped to a single instance. Animation is first applied to the *tet*Mesh and then the corresponding TLAS over the instances (*tet*LAS) is updated.

Current real-time ray tracing APIs, such as DXR [Microsoft 2020], do not have native support for tetrahedral primitives. In the following, we will illustrate how to map the two variants of our method (see Section 4.1 and Section 5.1) to the existing DXR API functionality.

In general, we start by placing a tetrahedral cage around each scene object in its original form (non-animated). The clipping preprocessing phase creates a *μMesh* per tetrahedron. These steps are the same for both methods. Similarly, the *μ*BLAS associated with a tetrahedron remains static and independent of the transformation applied. Depending on the variant, the animation transformation is either integrated into the instance transformation or directly applied to the tetrahedral cage. If the same object is animated with different transformations applied, we only need to replicate the tetrahedra cage, which reduces memory consumption (see Section 7). An example of this new hierarchy that reuses the *μ*BLAS is shown in Figure 8.

Instance Transformation-Based Variant. For the instance-based variant, each *μ*BLAS can be directly mapped to a DXR BLAS built over triangles. The ray-triangle intersections are handled automatically by the DXR intersection pipeline. Now, tetrahedra of all animated objects, each having its own tetrahedral cage, are put together into a single combined list. Each tetrahedron in this list is mapped to a single DXR instance. Each DXR instance transformation matrix is set to the tetrahedron's transformation as described in Section 4.1. In detail, we first precompute the transformation matrices $(M_i^{-1})_{3x4}$ (see Equation 1) for all tetrahedra. These matrices are static during rendering. A compute shader is dispatched to animate the vertices of all tetrahedra cages. Another compute shader reads these vertices to compute matrices A_i (see Equation 2). These matrices are combined into a single matrix O_i per tetrahedron. The 3×4 DXR instance transform matrix is set to the combined matrix $O_i A_i (M_i^{-1})_{3x4}$. Note that shading normals need to be transformed by the inverse transpose of the instance transforms. This matrix is computed once per frame for each animated tetrahedron.

Watertight Variant. For the watertight variant, we cannot use a DXR BLAS built over the triangles of the *μMesh*, as the triangle vertices and the bounding volumes are stored as 4D barycentric coordinates relative to a given tetrahedron (see Section 5.1), and DXR does not offer direct support for this. The reconstruction of the 3D world-space vertices and bounding volumes from relative 4D barycentric encodings has to be computed manually in software. We therefore leverage the DXR procedural geometry support. Each time a ray enters a tetrahedron, a procedural shader that traverses a precomputed binary 4D BVH is executed. The 4D position data is used to reconstruct the 3D world-space positions of two nodes' AABBs during ray traversal as well as the three triangle vertices (see Section 5.1) for the ray-triangle intersection test. A TLAS is built over the AABBs of the animated tetrahedra, with a one-to-one mapping between instance and tetrahedron.

7 Results

For our evaluation, we implemented a tetrahedra-based ray tracing framework, written in DirectX 12 and HLSL. All experiments were conducted on an AMD Radeon RX 9070 XT GPU with 16 GB of memory, running Windows 11. Render timings include tracing two rays per pixel (one primary and one secondary shadow ray) as well as shading and sampling. BLAS objects are updated using the *ACCELERATION_STRUCTURE_BUILD_FLAG_ALLOW_UPDATE* and *PREFER_FAST_BUILD* flags, while the TLAS is rebuilt with PREFER_FAST_TRACE.

We tested our approach with three different scenes: *Trees*, *Grass Patches*, and *Frogs* (see Figure 9), each varying in complexity and the count of animated objects. For each scene, we replicate the base object multiple times and assign a unique tetrahedral animation cage to each replica. Each base object was tested with medium and high tetrahedral cage resolutions. Both resolutions provide sufficient accuracy, so that no visible differences compared to animating all triangles along the chosen camera path were observed.

Adapting Bone-Based Animation. The *Frog* model is the only test scene that supports bone-based animation, which needs to be adapted to the tetrahedral cage. For each vertex of the tetrahedral

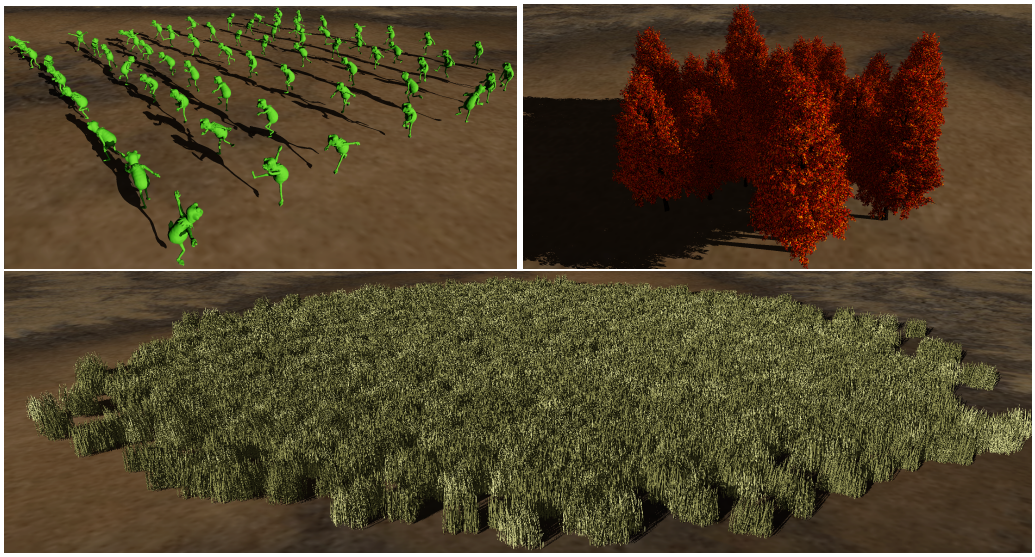

Fig. 9. Test scenes: (left) *Frogs*: set of bone animated characters with 26.1m triangles, each *Frog* object has 249k vertices and 408k triangles, (right) *Trees*: a group of animated trees with 39.5m triangles, each *Tree* object has 1.38m vertices and 1.58m triangles, (bottom) *Grass Patches* with 479m triangles, each *Grass Patch* object has 60k vertices and 120k triangles.

cage, we find the closest vertex of the original triangle model and use its bone IDs and bone weights. Note that this simple approach could be improved by a more advanced choice of bone IDs and weights, or through adaptation to the structure of the underlying skeleton. Taking bone animation into account, we extend our triangle clipping algorithm to deal with the per-vertex bone data. We collect up to 12 unique bone IDs per triangle and extend each vertex to carry the associated bone IDs and weights. If a vertex did not originally contain a particular bone ID, we set the weight for this bone to zero. We can clip the weights of all bones in a triangle like other vertex attributes. After the triangle clipping is done, we sort the vertex bone IDs of the vertices of the resulting triangles by bone weight. We keep the four largest weights along with their associated bone IDs and normalize the weights of these remaining four bones. Note that applying bone animation to the tetrahedral cage only approximates the bone-based animation of the original triangle model. The quality of the animation depends on the resolution of the tetrahedral cage. Note that during shading, we fetch the three non-deformed vertex normals of the triangle hit and skin those to mimic the behavior of the skinned original geometry.

Clipping. Our method requires that the original triangles of an object are clipped against the *tet*Mesh, which introduces additional vertices and triangles (see Table 1). The increase depends on the resolution of the tetrahedral cage and, for our test objects, is between 1.3 and 2.3× for triangles and between 1.4 and 3.7× for vertices. A higher number of triangles and vertices similarly affects the size of μBLASes. However, since μBLASes are reused for different animations of the same object, the initially higher memory consumption is quickly amortized as the reuse count increases.

Memory and Performance Comparisons. First, we compare memory consumption and rendering times between our watertight and non-watertight tetrahedra-based methods (see Table 2). Although memory consumption of the watertight method is only 2.3 – 3.2× that of the non-watertight method,

Table 1. Statistics for a single animated test object. The number of triangles increases by 1.3 – 2.3× after clipping. The average number of triangles per tetrahedron ranges from a few hundred to a few thousand, depending on the scene and initial voxel resolution. The number of tetrahedra per object is two to three orders of magnitude smaller than the number of triangles, which significantly reduces the animation cost.

Voxel Res	#Tets #Vtx	#Tris #Vtx Original	#Tris #Vtx Clipped	#Tris/Tet Avg Min Max
Tree				
5 × 8 × 5	0.64k 0.204k	1.58m 1.38m	2.15m (1.36×) 1.96m (1.42×)	3.3k 1 13k
9 × 12 × 9	2.32k 0.639k	1.58m 1.38m	2.55m (1.61×) 2.35m (1.70×)	1.1k 1 4.3k
Grass Patch				
6 × 5 × 6	0.66k 0.659k	120k 60.14k	207k (1.73×) 149k (2.48×)	0.3k 1 2k
10 × 9 × 10	2.42k 0.722k	120k 60.14k	277k (2.31×) 223k (3.70×)	0.1k 1 0.9k
Frog				
15 × 15 × 15	1.91k 0.711k	408k 249k	618k (1.51×) 418k (1.68×)	0.3k 1 1.7k
25 × 25 × 25	5.55k 19.24k	408k 249k	761k (1.87×) 569k (2.29×)	0.1k 1 0.8k

Table 2. Memory consumption and render time of our non-watertight and watertight methods for 25 *Trees*, 500 *Grass Patches*, and 81 *Frogs*. Our watertight method requires 2.3 – 3.2× more memory than non-watertight. The non-hardware-accelerated software fallback introduces a 19 – 80× render time overhead.

	25 Trees		*500 Grass Patches*		*9×9 Frogs*	
	Mem	Render Time	Mem	Render Time	Mem	Render Time
Non-Watertight	209.7 MB	2.96 ms	145.6 MB	1.35 ms	190 MB	1 ms
Watertight	540.5 MB	203 ms	459.4 MB	108 ms	442 MB	19 ms
Overhead	2.5×	68.6×	3.2×	80×	2.3×	19×

the required software fallback introduces a significant runtime overhead of 19 – 80×, making it unsuitable for real-time rendering and more applicable as a reference solution. This makes only the non-watertight method applicable to real-time rendering on current generation GPU hardware, while the watertight version can still be used as a validation reference.

Next, we compare our non-watertight tetrahedra-based method against the *standard approach*, which animates each individual triangle and then updates the triangle-based BLAS. Figure 10 shows a detailed breakdown of memory consumption, as well as animation, acceleration structure build, and render times for increasing numbers of *Grass Patch* objects. With an increasing number of animated triangles, the standard approach exhibits steeper linear scaling compared to our approach. By comparison, our approach is nearly flat, as the number of tetrahedra is two to three orders of magnitude smaller, resulting in a 16.1× smaller memory consumption. The same applies to animation and acceleration structure build times. The render time of our non-watertight version is slightly higher than for the standard approach. For ray tracing-based applications, the total time, which includes animation, acceleration structure updates, and rendering, is crucial. Here, our non-watertight approach outperforms the standard approach by up to 9×, while providing similar visual animation quality.

Figure 11 shows similar results for the *Trees* and *Frogs* scenes. With an increasing number of objects, the standard approach shows a steep increase in memory consumption, acceleration structure update time, and render time. Even for a high number of objects per scene, our approach shows only a marginal increase in memory consumption and update time, which results in a lower

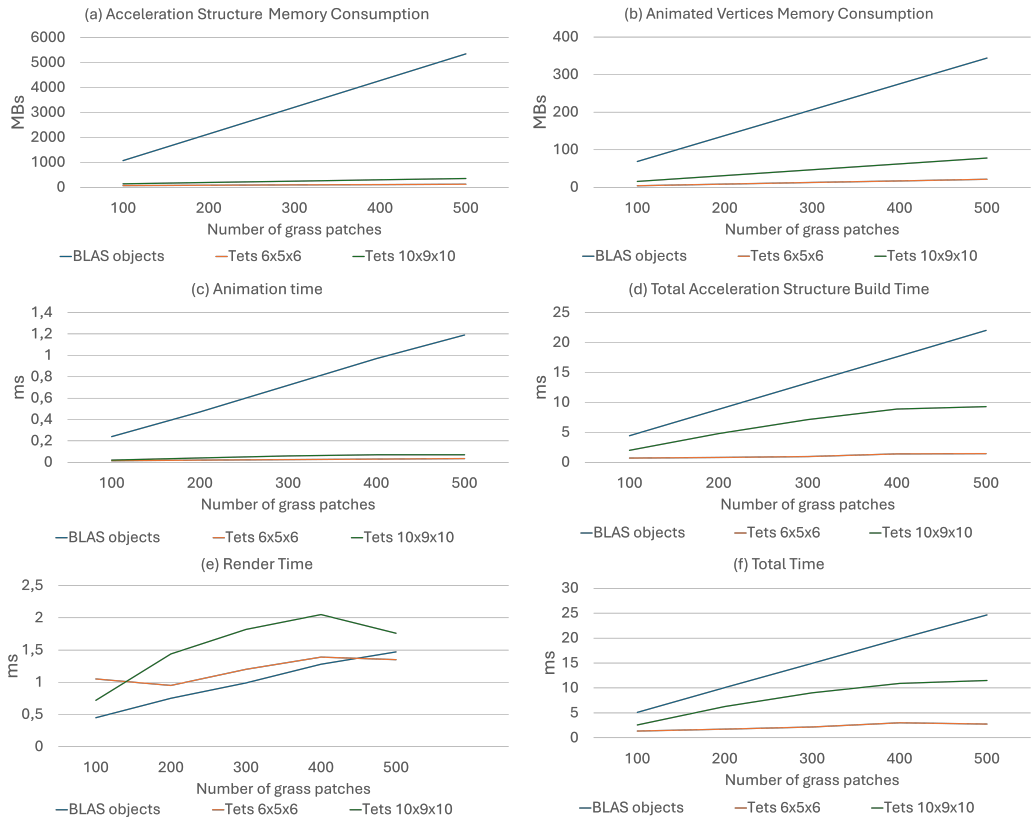

Fig. 10. Comparison of memory consumption, animation, acceleration structure build, and render times between the standard approach (*BLAS objects*) and our non-watertight tetrahedra-based approach, for an increasing number of *Grass Patch* objects (see Figure 9). Two tetrahedral cage resolutions are used: medium ($6\times5\times6$) and high ($10\times9\times10$). The standard approach animates each individual triangle and then updates the triangle-based BLAS. With an increasing number of patches, this results in steep linear increase in memory consumption (*(a)* and *(b)*) as well as animation *(c)*, and acceleration structure build times *(d)*. By comparison, our non-watertight tetrahedra-based approach only exhibits a marginal increase. The *render time (e)* of our approach and the standard approach are very similar. For 500 grass patches, the *total time (f)*, including animation, acceleration structure, and render time, our non-watertight approach outperforms the standard approach by up to 9×, depending on the resolution of the tetrahedral cage.

total time. Finally, we combined all three example scenes into a single scene (see Figure 1) which cannot be ray traced with the standard approach due to the GPU memory limit of 16 GB. Our approach is able to render this complex animated scene at ~60 fps (two rays per pixel). Table 3 shows a detailed timing breakdown. With around 2.5m tetrahedra for this scene, the most expensive phases per frame are animating the *tet*Mesh vertices and rebuilding the TLAS over all tetrahedra.

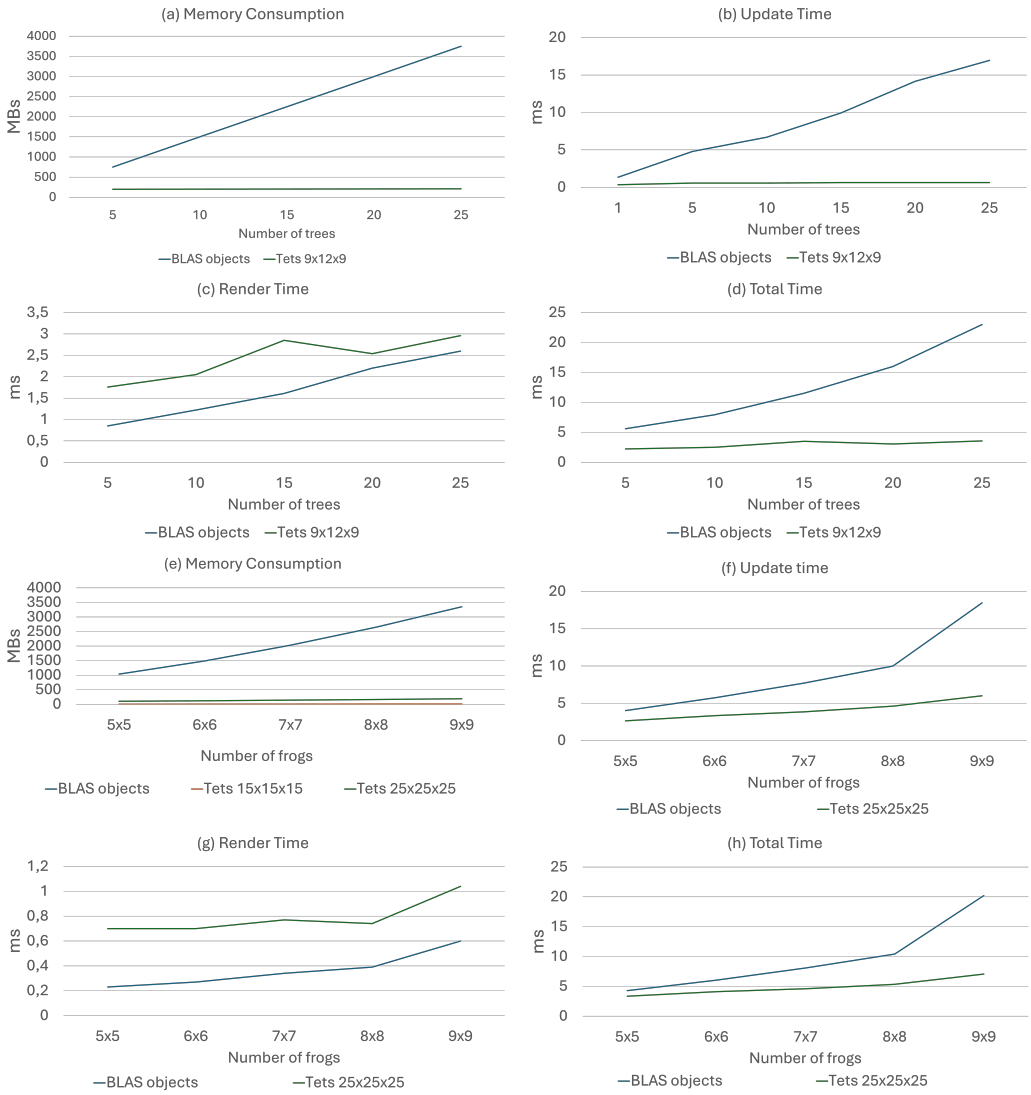

Fig. 11. Memory consumption (vertices and BLASes), update time (vertex animation and acceleration structure), render and total time for an increasing number of *Tree* objects and *Frogs*. Similar to the *Grass Patch* scene (see Figure 10), the standard approach shows a steep increase in memory consumption, update and total times, while our approach remains almost flat.

Table 3. Timing breakdown for the combined scene (see Figure 1). Only our tetrahedra-based approach is able to render this scene at ~ 60 fps (12.43 ms/frame) while only consuming 770.10 MB of GPU memory. The *update* time, which includes animating the *tet*Mesh vertices and rebuilding the TLAS (*tet*LAS) over the tetrahedra, is the most expensive step and consumes 78% of the total time per frame.

Anim Triangles	Tets	Anim Time	Update Time	Render Time	Total Time
584m	2.8m	0.35 ms (3%)	9.66 ms (78%)	2.42 ms (19%)	12.43 ms

8 Limitations

Our method approximates connectivity-preserving triangle mesh animation using a piecewise-linear deformation induced by the tetrahedral cage. Because each tetrahedron induces an affine deformation, geometry inside that tetrahedron is restricted to a locally linear motion. This approximation is well suited to the following scenarios:

- Dense, uniquely-animated geometry with fixed triangle connectivity.
- Smooth deformations such as bending of foliage, grass, or trees, as well as crowd motion.
- Skeletal/bone animation, provided that the cage resolution is sufficiently high to approximate skinning.

The method is less suited for:

- Substantial changes to mesh topology (e.g., explosions, destruction, cutting, etc.).
- Per-triangle or high-frequency motion below the cage scale (e.g., cloth or facial animation with folds smaller than the cage resolution).
- Complex character skinning with sharp weight changes near joints.
- Rigid objects, for which conventional instancing is typically simpler and often preferable.
- Keyframe-based animation, in which vertices are animated by interpolating their positions across keyframes.

Artifacts arising from insufficient cage resolution can often be reduced by locally refining the tetrahedral cage (see Figure 12). Supporting keyframe-based animation requires recomputing the transformation between corresponding triangle vertices in successive keyframes and applying it to the tetrahedral cage. For animations that require per-triangle control or that alter triangle connectivity, the affected μBLASes would need to be rebuilt. Bone-animated objects additionally require assigning bones and weights to the tetrahedral cage.

Finally, the method impacts ray tracing performance by trading increased ray traversal and intersection costs for substantially reduced animation and acceleration-structure update costs. The instance-based variant requires slight geometric overlap at shared tetrahedron boundaries to mitigate watertightness artifacts, whereas the watertight variant relies on an expensive software fallback. Because each original triangle is clipped against the *tet*Mesh, the resulting sub-triangles may be distributed across multiple tetrahedra. Consequently, when a ray intersects adjacent tetrahedra, a shared edge may be tested multiple times, leading to repeated intersections of the same original primitive. Assigning unique ownership of shared edges to tetrahedra can help mitigate this issue.

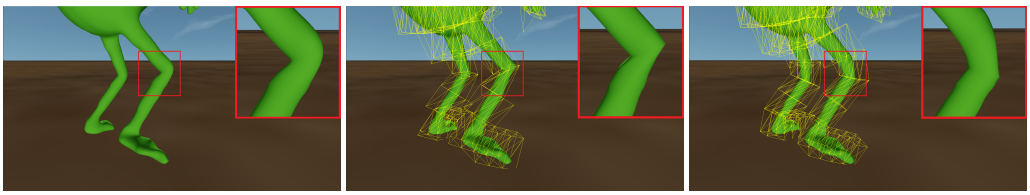

Fig. 12. (Left) Original skinned triangle model. (Middle) Animation artifacts near joints caused by insufficient tetrahedral cage resolution. (Right) A higher tetrahedral cage resolution around the targeted area reduces the artifacts.

9 Conclusion and Future Work

In this paper, we present a method for efficiently ray tracing massive amounts of uniquely animated geometry. Our method decouples the complexity of animating triangular geometry from the geometry's triangle density. Our approach requires only a *tet*AnimCopy per uniquely animated object, thereby dramatically reducing memory consumption, acceleration structure update times, and the cost of object animation itself. Applications that are limited by acceleration structure build times, animation times, or memory footprint when animating massive amounts of geometry can directly benefit from our approach.

Our approach supports varying levels of detail (LOD) by using tetrahedral cages at different resolutions. Higher-resolution tetrahedral cages can improve the quality of animation results at the expense of increased memory consumption and acceleration structure update times. In general, our method can be directly combined with existing solutions for per-object LOD, staggered acceleration structure updates, and the use of cluster-based geometry representations or TLAS partitions [Microsoft 2026]. It is also possible to mix tetrahedral cage-based animation with standard mesh-based or rigid-body animation. Direct hardware and API support for tetrahedra would improve culling efficiency; similarly, support for multi-level instancing would allow for more flexibility in trading ray tracing time versus acceleration structure build times.

Future work will focus on improving the voxel-to-tetrahedra algorithm and the quality of bone-based object animation, e.g., by running an optimization process over all keyframes to find the set of bones and bone weights that minimize the difference between the skinned vertices of the high-resolution object and the vertex positions produced by animated tetrahedral cages. Further analysis of the curvature of the object's surface around each vertex of each keyframe could be used to improve the animation quality. If a tetrahedron contains vertices with a minimum curvature above a threshold, the preprocessing stage could split the tetrahedron into sub-tetrahedra and continue recursively.

Acknowledgments

Model courtesy: *Frog* (Sketchfab.com), *Tree* (cgtrader.com), *Grass Patch* (cgtrader.com). The authors would like to thank Jakub Bokšanský, Joshua Barczak, Andrew Kensler, Nathan Morrical, Dave Oldcorn and the anonymous reviewers for their feedback.

References

Pierre Alliez, David Cohen-Steiner, Mariette Yvinec, and Mathieu Desbrun. 2005. Variational tetrahedral meshing. *ACM Transactions on Graphics (TOG)* 24, 3 (2005), 617–625.

Aytek Aman, Serkan Demirci, and Uğur Güdükbay. 2022. Compact tetrahedralization-based acceleration structures for ray tracing. *J. Vis.* 25, 5 (2022), 1103–1115.

Carsten Benthin and Christoph Peters. 2023. Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail. *Proceedings of High Performance Graphics* 42, 8 (2023).

Brian Karis. 2022. Journey to Nanite. https://www.highperformancegraphics.org/slides22/Journey_to_Nanite.pdf

Vid Domiter and Borut Žalik. 2008. A sweep-line algorithm for triangulation of figures. *Computers & Geosciences* 34, 11 (2008), 1551–1561.

Hans Freudenthal. 1942. Simplizialzerlegungen von Beschränkter Flachheit. *Annals of Mathematics* 43, 3 (1942), 580–582.

Holger Gruen, Carsten Benthin, Andrew Kensler, Joshua Barczak, and David McAllister. 2024. Ray Tracing Animated Displaced Micro-Meshes. *Comput. Graph. Forum* 43, 7 (2024).

Warren Hunt, William Mark, and Don Fussell. 2007. Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing*. 47–54.

Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. 2011. Bounded biharmonic weights for real-time deformation. *ACM Transactions on Graphics (TOG)* 30, 4 (2011), 1–8.

Pushkar Joshi, Mark Meyer, Tony DeRose, Brian Green, and Tom Sanocki. 2007. Harmonic coordinates for character articulation. *ACM Transactions on Graphics (TOG)* 26, 3 (2007), 71–es.

Marek Kácerik and Jiří Bittner. 2025. UBVH: Unified Bounding Volume and Scene Geometry Representation for Ray Tracing. *Proceedings of High Performance Graphics* 44, 8 (2025).

Tero Karras and Timo Aila. 2013. Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics*. 89–100.

Andrew Kensler. 2008. Tree Rotations for Improving Bounding Volume Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing*. 73–76.

Khronos Group. 2020. Vulkan Ray Tracing Extensions Specification. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_ray_tracing.html

Khronos Group. 2024. *Vulkan Extension: VK_NV_partitioned_acceleration_structure.* The Khronos Group Inc. https://docs.vulkan.org/features/latest/features/proposals/VK_NV_partitioned_acceleration_structure.html

Ares Lagae and Philip Dutré. 2008. Accelerating Ray Tracing using Constrained Tetrahedralizations. *Computer Graphics Forum* 27 (2008).

Won-Jong Lee and Gabor Liktor. 2020. Lazy Build of Acceleration Structures with Traversal Shaders. In *Proceedings of ACM SIGGRAPH Asia (Technical Communication)*.

Pacôme Luton and Thibault Tricard. 2025. Real-time rendering of animated meshless representations. In *HPG 2025 - Conference on High Performance Graphics*. Eurographics.

Andrea Maggiordomo, Henry Moreton, and Marco Tarini. 2023. Micro-mesh construction. *ACM Transactions on Graphics (TOG)* 42, 4 (2023).

Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiří Bittner. 2021. A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum* 40, 2 (2021), 683–712.

Johannes Mezger, Stefan Kimmerle, and Olaf Etzmuß. 2003. Hierarchical techniques in collision detection for cloth animation. In *Journal of WSCG*, Vol. 11. 322–329.

Microsoft. 2020. DirectX Raytracing (DXR) Spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

Microsoft. 2026. *DirectX Raytracing (DXR) Spec, Part 2.* https://microsoft.github.io/DirectX-Specs/d3d/Raytracing2.html

Nate Morrical, Ingo Wald, and Valerio Pascucci. 2023. RTX-Accelerated Volumetric Rendering using Tetrahedral Meshes. In *Proceedings of High Performance Graphics 2023*. ACM.

Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. 2005. Real time physics: class notes. *ACM SIGGRAPH 2005 Courses* (2005), 1–90.

Matthias Nießner. 2013. *Rendering Subdivision Surfaces using Hardware Tessellation.* Ph. D. Dissertation. University of Erlangen-Nuremberg. In collaboration with Stanford University.

NVIDIA Corporation. 2024. *NVIDIA RTX Mega Geometry.* NVIDIA Developer. https://developer.nvidia.com/blog/nvidia-rtx-mega-geometry-now-available-with-new-vulkan-samples/

Hang Si. 2006. *A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator Version 1.4 User's Manual.* https://wias-berlin.de/software/tetgen/files/tetgen-manual.pdf

Bowen Wang, Yingjie Sun, Nengxiong Xu, and Gang Mei. 2020. A Clustering-Based Bubble Method for Generating High-Quality Tetrahedral Meshes of Geological Models. *Applied Sciences* 10 (2020).

Jun Wang and Zeyun Yu. 2012. Feature-sensitive tetrahedral mesh generation with guaranteed quality. *Computer-Aided Design* 44 (2012).

Kaixin Yu, Yifu Wang, Peng Song, Xiangqiao Meng, Ying He, and Jianjun Chen. 2024. Weighted Squared Volume Minimization (WSVM) for Generating Uniform Tetrahedral Meshes. *arXiv* (2024).

A 4D Barycentric Space to 3D World-Space BVH Upholds Bounding Property

We must show that if a point $\mathbf{P}$ expressed by a set of barycentric coordinates is inside the tetrahedron, then $\mathbf{P}$ will always be contained within the 3D AABB generated from the 4D barycentric bounds, regardless of how the tetrahedron vertices $\mathbf{V}_i$ change.

Any point $\mathbf{P}$ in 3D space within a tetrahedron can be expressed as:

$$\mathbf{P} = \sum_{i=1}^{4} \lambda_i \mathbf{V}_i$$

where $\sum \lambda_i = 1$ and $\lambda_i \geq 0$. In our 4D BVH, a node (representing a collection of triangles) defines a 4D hyper-tetrahedron B_{4D} such that for every point $\mathbf{P}$ in that subtree, its coordinates satisfy:

$$\lambda_{i,\min} \leq \lambda_i \leq \lambda_{i,\max} \quad \text{for } i \in \{1, 2, 3, 4\}$$

When the tetrahedron deforms, the vertices $\mathbf{V}_i$ change to $\mathbf{V}'_i$. The new position of the point is:

$$\mathbf{P}' = \sum_{i=1}^{4} \lambda_i \mathbf{V}'_i$$

Our goal is to prove that $\mathbf{P}'$ is contained in the projected 3D box $B'_{3D} = [\mathbf{x}'_{\min}, \mathbf{x}'_{\max}]$. The algorithm calculates the x-axis bounds (and similarly for y, z) as:

$$x'_{\min} = \sum_{i=1}^{4} \min(\lambda_{i,\min} V'_{i,x}, \lambda_{i,\max} V'_{i,x}) \qquad x'_{\max} = \sum_{i=1}^{4} \max(\lambda_{i,\min} V'_{i,x}, \lambda_{i,\max} V'_{i,x})$$

Let L_i be the interval $[\lambda_{i,\min}, \lambda_{i,\max}]$. Since $\lambda_i \in L_i$, we are looking at the properties of the linear combination of intervals. For a single term $T_i = \lambda_i V'_{i,x}$, the range of possible values is:

$$\text{range}(T_i) = \{\lambda_i V'_{i,x} \mid \lambda_i \in [\lambda_{i,\min}, \lambda_{i,\max}]\}$$

Because this is a linear function of λ_i, the minimum and maximum of T_i must occur at the boundaries of the interval L_i. If $V'_{i,x} \geq 0$, the minimum is $\lambda_{i,\min} V'_{i,x}$ and the maximum is $\lambda_{i,\max} V'_{i,x}$. If $V'_{i,x} < 0$, the minimum is $\lambda_{i,\max} V'_{i,x}$ and the maximum is $\lambda_{i,\min} V'_{i,x}$. In both cases, the contribution of the i-th vertex to the total x-coordinate is bounded by:

$$\min(\lambda_{i,\min} V'_{i,x}, \lambda_{i,\max} V'_{i,x}) \leq \lambda_i V'_{i,x} \leq \max(\lambda_{i,\min} V'_{i,x}, \lambda_{i,\max} V'_{i,x})$$

By the Subdistributivity Property of interval arithmetic, the sum of these individual bounds is a guaranteed enclosure for the sum of the terms:

$$\sum_{i=1}^{4} \min(\lambda_{i,\min} V'_{i,x}, \lambda_{i,\max} V'_{i,x}) \leq \sum_{i=1}^{4} \lambda_i V'_{i,x} \leq \sum_{i=1}^{4} \max(\lambda_{i,\min} V'_{i,x}, \lambda_{i,\max} V'_{i,x})$$

Substituting the definitions of $x'_{\min}$, P'_x, and $x'_{\max}$:

$$x'_{\min} \leq P'_x \leq x'_{\max}$$

The same logic is applied to the y and z axes, hence every point $\mathbf{P}'$ whose barycentric coordinates are within the 4D bounds is guaranteed to be inside the projected 3D AABB. Even if the tetrahedron is inverted, scaled to zero, or mirrored, the min/max logic handles the sign changes of V'_i automatically.