J Vis (2022) 25:1103–1115
https://doi.org/10.1007/s12650-022-00842-x

REGULAR PAPER

Aytek Aman · Serkan Demirci · Uğur Güdükbay

Compact tetrahedralization-based acceleration structures for ray tracing

Received: 27 September 2021 / Revised: 12 March 2022 / Accepted: 3 April 2022 / Published online: 12 May 2022

Abstract We propose compact and efficient tetrahedral mesh representations to improve the ray-tracing performance. We reorder tetrahedral mesh data using a space-filling curve to improve cache locality. Most importantly, we propose efficient ray-traversal algorithms. We provide details of the regular ray-tracing operations on tetrahedral meshes and the GPU implementation of our traversal method. We demonstrate our findings through a set of comprehensive experiments. Our method outperforms existing tetrahedral mesh-based traversal methods and yields comparable results to the traversal methods based on the state-of-the-art acceleration structures such as *k*-dimensional (*k*-d) tree and Bounding Volume Hierarchy (BVH) in terms of speed. Storage-wise, our method uses less memory than its tetrahedral mesh-based counterparts, thus allowing larger scenes to be rendered on the GPU.

Keywords Ray tracing · Ray-surface intersection · Acceleration structure · Tetrahedral mesh · Bounding volume hierarchy (BVH) · *k*-dimensional (*k*-d) tree

1 Introduction

The most common approach to speed up ray-surface intersection calculations in ray tracing is to use spatial subdivision structures that partition the scene to enclose the polygons in different volumes. Ray-tracing algorithms can avoid ray-triangle intersection tests if the enclosing volume for a triangle does not intersect with the ray. Popular acceleration structures are regular grids, octrees, Bounding Volume Hierarchy (BVH), and *k*-dimensional (*k*-d) tree. BVH and *k*-d tree are the most preferred acceleration structures for ray tracing, thanks to the recent advancements in the construction and traversal methods.

A recent alternative to accelerate ray-surface intersection calculations is to use tetrahedralizations. A tetrahedral mesh is a three-dimensional (3D) structure that partitions the 3D space into tetrahedra. Constrained tetrahedralizations are a special case of tetrahedralizations that take the input geometry into account. In the resulting mesh, the components of the input geometry such as faces, line segments, and points are preserved. Similar to their two-dimensional (2D) counterparts, tetrahedralizations can be

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s12650-022-00842-x.

A. Aman · S. Demirci · U. Güdükbay (✉)
Department of Computer Engineering, Bilkent University, 06800 Ankara, Turkey
E-mail: gudukbay@cs.bilkent.edu.tr

A. Aman
E-mail: aytek.aman@cs.bilkent.edu.tr

S. Demirci
E-mail: serkan.demirci@bilkent.edu.tr

constructed in such a way that they exhibit Delaunay property, i.e., tetrahedra are close to regular. There are three types of constrained tetrahedralizations: *Conforming Delaunay Tetrahedralization*, *Constrained Delaunay Tetrahedralization*, and *Quality Delaunay Tetrahedralization* (Lagae and Dutré 2008a).

Lagae and Dutré (2008a) use constrained tetrahedral meshes for rendering typical 3D scenes. They tetrahedralize the space between objects in a constrained manner where the triangles in the scene geometry align with the triangles of the tetrahedral mesh. Then, they calculate ray-triangle intersections by traversing the tetrahedral mesh. Because a tetrahedral mesh is not a hierarchical structure, ray-surface intersections are mostly calculated by traversing a few tetrahedra. Besides, this approach has the advantages of providing a unified data structure for global illumination, handling deforming geometry if the topology (connectivity) of the mesh does not change, easily applying level-of-detail approaches, and ray tracing on the Graphics Processing Unit (GPU). Despite these advantages, the state-of-the-art traversal methods for tetrahedral meshes, such as Scalar Triple Product (ScTP), are still a magnitude or two slower than the *k*-d tree-based traversal, as Lagae and Dutré (2008a) state. We aim to improve the performance of the tetrahedral mesh-based traversal for ray tracing as follows.

- We propose a compact tetrahedral mesh representation to improve cache locality and to utilize memory alignment.
- We sort tetrahedral mesh data (tetrahedra and points) using a space-filling curve to improve cache locality.
- We propose an efficient tetrahedral mesh traversal algorithm using a modified basis that reduces the cost of point projection, which is frequently used during traversal.
- We utilize the GPU to speed up the ray-surface intersection calculations.

We also propose a technique to associate vertex attributes (normals, texture coordinates, and so on) with the tetrahedral mesh data. Through experiments, we observe that our method performs better than the existing tetrahedral mesh-based traversal methods in terms of the computational cost. In certain scenes, especially the scenes with challenging geometry where there are long, extended triangles, we observe a better rendering performance than the *k*-d tree and BVH implementations of the *pbrt-v3* (Pharr et al. 2016). Although this method cannot replace and improve upon the state-of-the-art accelerators (such as BVH and *k*-d tree) because of its disadvantages in its current form, its orthogonal strengths compared to the alternatives make it valuable and promising. This is especially important for aggregate structures where accelerators with different advantages can be combined to have the best of both worlds.

2 Related work

2.1 Acceleration structures

First proposed by Fujimoto et al. (1988), a regular grid partitions the 3D scene into equally sized boxes where each box keeps a list of triangles. During traversal, some well-known algorithms such as the three-dimensional digital differential analyzer (3D DDA) can be used to quickly determine the boxes that intersect with the ray. Although there are compact and robust acceleration structures such as the one proposed in Lagae and Dutré (2008b), one major disadvantage of the regular grid is its non-adaptive structure. The majority of the grid cells may not contain any triangles, while some grid cells may have a large number of triangles, which increases the average traversal cost.

One of the popular structures in the literature is BVHs. A BVH is a collection of hierarchical bounding volumes that enclose the objects in the scene. BVHs improve the ray-tracing performance by culling the scene geometry using bounding volume intersection tests. Therefore, less triangle-ray intersection tests have to be performed compared to the brute force full scene traversal. Modern BVH construction techniques employ Surface Area Heuristic (SAH) (Goldsmith and Salmon 1987) to construct acceleration structures that perform well. The state-of-the-art BVHs are constructed using a greedy top-down plane-sweeping algorithm proposed by Goldsmith et al (1990), which is extended by Stitch et al. (2009) using spatial splits. Wodniok et al. (2017) use recursive SAH values of temporarily constructed SAH-built BVHs to reduce ray-traversal cost further.

The octree is another spatial indexing structure that is used to accelerate ray tracing (Glassner 1984). It divides the space into eight subspaces in a recursive manner. During ray tracing, the octree is used to index the scene into subspaces and it is useful to determine the subspaces that intersect with the rays. After an octree is constructed, triangles that reside in these subspaces can be queried and the closest intersection with

the rays can be found by performing a relatively small number of ray-surface intersection tests compared to the brute force approach.

Similar to the octree, the k-d tree is also a space partitioning structure that divides the space into two subspaces at each level by alternating the split axis. To reduce the average ray-traversal cost on a k-d tree, these split planes are selected using the SAH, which is proposed by MacDonald and Booth (1990). SAH-based k-d tree construction approaches are later improved by Havran and Bittner (2002). Wald et al. (2006) propose a SAH-based k-d tree construction scheme with $\mathcal{O}(N \log N)$ computational complexity. The k-d trees constructed using the SAH are adaptive to the scene geometry. This means that if a ray is not in the proximity of any scene geometry, only a few tree nodes are traversed. This reduces the computation cost of ray tracing on scenes where primitives in the scene are not uniformly distributed, which is a common scenario for 3D scenes.

In many ray-tracing applications, rays share a common point such as rays that originate from the camera or rays that are cast to the light sources after ray-surface intersections. The structures that are discussed above do not exploit the characteristics of such rays in ray tracing. There exist better structures that take advantage of rays that share a common point in space and creates indices accordingly. Light Buffer (Haines and Greenberg 1986) is an approach that partitions the scene according to one light source in the scene, which is then used for shadow testing. Hunt et al. (2008) propose the perspective tree, which is similar to the Light Buffer that uses a 3D grid in a perspective space considering the position of the light source or the camera as root.

2.2 Tetrahedral mesh construction and traversal

Given an input geometry, a tetrahedral mesh can be constructed using well-known algorithms in computational geometry. TetGen (Si 2015) is a commonly used tool to generate tetrahedral meshes. TetGen uses Bowyer-Watson algorithm (Bowyer 1981; Watson 1981) and the incremental flip (Edelsbrunner and Shah 1992) algorithms. Both methods have the worst-case complexity of $\mathcal{O}(N^2)$. If points are uniformly distributed in space, the expected run-time complexity is $\mathcal{O}(N \log N)$. To ensure numerical robustness, robust geometric predicates are used (Shewchuk 1996).

There are tetrahedral mesh-based traversal methods used for accelerating ray-surface intersection calculations. Lagae and Dutré (2008a) use ScTP to traverse the tetrahedral mesh. Their method requires the computation of three to six ScTP to determine the exit face. ScTP computation involves a cross-product followed by a dot product on 3D vectors. Maria et al. (2017b) propose a fast tetrahedral mesh traversal method, which uses an efficient exit face determination algorithm based on Plücker coordinates.

Our method uses an efficient traversal method that works in 2D, resulting in very few floating-point operations per tetrahedron compared to these alternatives. Our data structures are also compact and memory aligned. We also use a space-filling curve to further improve cache locality. Maria et al. (2017a; 2014) also propose a new acceleration structure for ray tracing, constrained convex space partition (CCSP), as an alternative to tetrahedral mesh-based acceleration structures. CCSP is more suitable for architectural environments because such a partitioning of a scene contains a smaller number of convex volumes, rather than a large number of tetrahedra.

2.3 Raycasting for direct volume rendering

Direct volume rendering methods for rendering irregular grids, mostly represented as unstructured tetrahedral meshes, rely on raycasting and the composition of shades of samples along the rays throughout the volume to calculate pixel colors. For example, Silva et al. (1996; 1997) use a sweeping plane first applied in the x-z plane, and then a sweeping line applied on the z-axis. They process these sweep lines further to render volumetric data stored as an irregular grid. Berk et al. (2003) focus on the usage of hybrid methods to utilize the strengths of image- and object-space methods. They rely on a next-cell operation for determining the next tetrahedron that the ray travels, as proposed by Koyamada et al. (1992).

Garrity (1990) uses a simple traversal method where the ray is intersected with tetrahedra faces and the closest intersection gives the exit face for the tetrahedron. Koyamada (1992) uses two (on average) point-in-triangle tests in 2D to determine the exit face. Riberio et al (2007) use a more compact data structure for reduced memory usage during traversal. They also utilize ray coherence to reduce run-time memory usage. Later on, they (Maximo et al. 2008) improved this method by providing a hardware implementation with additional arrangements of the data structure for reduced memory usage. Marmitt and Slusallek (2006) use a

method proposed by Platis and Theoharis (2003), which employs Plücker coordinates of the ray and the tetrahedron edges to determine the exit face. They use the entry face information to reduce the number of tests to determine the exit face. They find the exit face using 2.67 ray-line orientation tests per tetrahedron on average.

Alternatively, cell trees (based on bounding interval geometry) Garth and Joy (2010) and tetrahedral trees Fellegara et al. (2020) provide efficient ways to answer point location queries in tetrahedral meshes thus can be used to speed up sampling operations in volumetric rendering. However, these techniques cannot be used efficiently to answer ray-triangle intersection queries. Additionally, these structures are not designed for tetrahedron traversal in a consecutive fashion.

We provide a fast and compact acceleration structure to quickly find ray-surface intersections for rendering 3D scenes composed of polygons (surface data). As opposed to direct volume visualization methods, our acceleration structure can handle queries for random rays scattered in different directions, given that their origin is already located (ray connectivity). Direct volume rendering techniques are geared toward rendering volumetric data from a specific camera position and orientation. In a recent work, our tetrahedral ray-traversal scheme is adapted to tetrahedron traversal (marching) consecutively for direct volume rendering methods for better cache utilization and reduced computational cost (Sahistan et al. 2021).

3 Tetrahedral mesh representation

We use a compact tetrahedral mesh representation for good cache utilization. We store tetrahedral mesh in two arrays as in Lagae and Dutré (2008a). The first array stores the point data, and the second array stores the tetrahedron data. Figure 1 depicts the tetrahedron data representation for typical scenarios.

Instead of using this representation, we propose three tetrahedron storage schemes that are more compact and better suited for efficient traversal: *Tet32*, *Tet20*, and *Tet16*, which are 32, 20 and 16 bytes, respectively. We store a common field, exclusive-or sum (xor-sum), in all these structures, inspired by xor linked list structures for reducing the memory requirements of linked lists (Sinha 2005). Mebarki (2018) uses a similar structure for compact 2D triangulations. VX^i denotes the xor-sum of the vertex indices of the i^{th} tetrahedron and V_j^i denotes the index of the j^{th} vertex of the i^{th} tetrahedron. We compute the xor-sum as follows.

$$VX^i = V_0^i \oplus V_1^i \oplus V_2^i \oplus V_3^i$$

Tet32 structure contains the first three vertex indices, xor-sum of all vertex indices, and four neighbor indices. Its memory layout is shown in Fig. 2.

With the *Tet32* representation, we can use the *xor* operation to quickly retrieve the index of the vertex that is not on a given face. We can get the index of the fourth point V_3^i as follows.

$$V_3^i = V_0^i \oplus V_1^i \oplus V_2^i \oplus VX^i.$$

This follows from the fact that xor operation is associative, commutative, and has the property $X \oplus X = 0$.

If the corresponding face is a part of the scene geometry, neighbor index data points to a structure, called *constrained face*. We use a single bitmask to identify such faces on the neighbor tetrahedron index field where the remaining 31 bits are used to reference either a neighboring tetrahedron or a constrained face depending on the value of the bitmask. Constrained face structure holds a reference to the actual triangle geometry and stores references to the neighboring two tetrahedra indices. These indices are used to recover and initialize the tetrahedron data when scattering rays are to be traced. It should be noted that multiple constrained faces can point to a single triangle when we allow triangles to be subdivided during tetrahedralization to enable high-quality tetrahedral meshes.

The xor-based storage scheme has the following advantages:

V_0^i	V_1^i	V_2^i	V_3^i
N_0^i	N_1^i	N_2^i	N_3^i

Fig. 1 Typical tetrahedron representation in the memory. V_j^i represents the index of the j'th vertex of the i'th tetrahedron. N_j^i represents the neighboring tetrahedron index, which is across the vertex V_j^i. Each field is an integer and four bytes long. Thus, the full tetrahedron data occupies 32 bytes of memory

V_0^i	V_1^i	V_2^i	VX^i
N_0^i	N_1^i	N_2^i	N_3^i

Fig. 2 *Tet32* structure. Each field is an integer and four bytes long. The tetrahedron data occupies 32 bytes of memory

- XOR-based representations are compact thus require less memory. Hence, the cache memory can be utilized more efficiently to reduce the rendering times. Additionally, more data can be fitted to the memory, especially critical for GPU implementations.
- Fetching the subsequent tetrahedron data becomes straightforward. Thanks to Xor representation, only one additional vertex needs to be fetched, and it can be fetched easily by XORing all the vertex indices making the traversal code more straightforward and efficient.

Our alternative tetrahedral representations, namely *Tet20* and *Tet16*, are even more efficient since their compact nature results in better cache utilization. These representations can be used in the exact same way during traversal since we can reconstruct full tetrahedron data given that traversal operations exhibit *ray connectivity*. However, both representations require additional operations to reconstruct tetrahedron data. These representations and operations required to reconstruct the tetrahedron data are explained in Appendix A.

4 Tetrahedron traversal

As the first step of tetrahedron traversal, we construct a 2D basis $b = (\mathbf{u}, \mathbf{v})$ from the ray direction using the method described in Duff et al. (2017). Then, we define a new 2D coordinate system C with basis b and origin o where r_o is the ray origin. We transform tetrahedron vertices to the coordinate system C to obtain four points in 2D. We determine the exit face in the initial tetrahedron using at most four points in triangle tests in 2D. The query point is at the origin because the ray origin is the center of the new coordinate system C. Once we determine the exit face, we keep the 2D coordinates and indices of the points of the exit face as p_0, p_1, p_2 and idx_0, idx_1, idx_2, respectively. We also fetch the next tetrahedron index using the neighbor data.

After the initialization step, we start traversing the tetrahedral mesh. We first fetch the index of the fourth corner of the next tetrahedron (three of them are already known because two neighboring tetrahedra share three vertices) using the following expression where XV^{next} denotes the xor sum of the next tetrahedron.

$$idx_3 = idx_0 \oplus idx_1 \oplus idx_2 \oplus XV^{next}$$

Using the index idx_3, we fetch the vertex from the points array, transform it to the new coordinate system, C, and use the resulting 2D point p_3 to decide the exit face of the ray (cf. Algorithm 1). Because the query point is at the origin after transformation, only four floating point multiplications and two floating point comparisons are sufficient. The exit face index is denoted as $exit\_face\_idx$ and resides across the point $p_{exit\_face\_idx}$ whose index is $idx_{exit\_face\_idx}$. To get the next tetrahedron, we use the $idx_{exit\_face\_idx}$ in the current tetrahedron data to fetch the corresponding neighbor tetrahedron index. Figure 3 illustrates the coordinate system transformation for a ray and a tetrahedron.

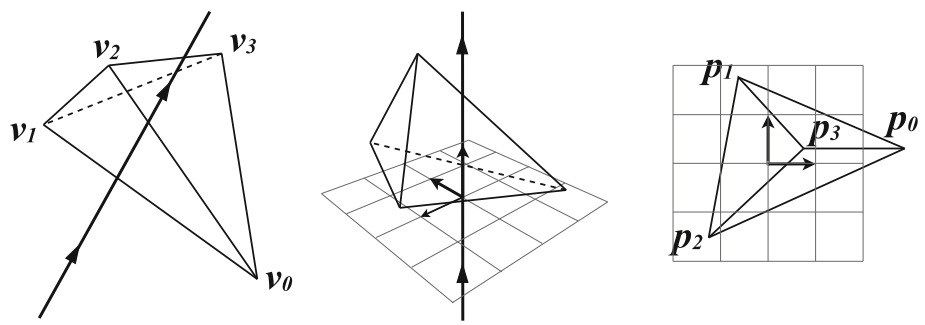


Fig. 3 Ray-tetrahedron intersection. Left: a ray and a tetrahedron. Middle: the tetrahedron transformed into the coordinate system defined by the ray. The ray coincides with the z-axis. Right: the tetrahedron projected onto 2D. The ray is at the origin and points to the viewer

Algorithm 1 Exit face selection

```
procedure GetExitFace(p_{0..3})
    exit_face_idx ← 0
    if det(p_3, p_0) < 0 then
        if det(p_3, p_2) ≥ 0 then
            exit_face_idx ← 1
        end if
    else if det(p_3, p_1) < 0 then
        exit_face_idx ← 2
    end if
    return exit_face_idx
end procedure
```

In *Tet32*, we simply search for $idx_{exit\_face\_idx}$ in the current tetrahedron. Since vertex and neighbor indices correspond to each other, location of the $idx_{exit\_face\_idx}$ (value from 0 to 3) also reveals the location of the neighbor to be traversed next. We describe this process in Algorithms 2 and 3.

Algorithm 2 Tetrahedron traversal loop for *Tet32*

```
while tet_idx ≥ 0 do
    idx_{exit_face_id} ← idx_3
    idx_3 ← idx_0 ⊕ idx_1 ⊕ idx_2 ⊕ VX^{tet_idx}
    v_new ← points_{idx_3} − r_o
    p_{exit_face_idx} ← p_3
    p_3 ← (u · v_new, v · v_new)
    exit_face_idx = GetExitFace(p_{0..3})
    next_tet_idx = GetNextTet32(tet_idx, idx_i)
end while
```

Algorithm 3 Next tetrahedron determination for *Tet32*

```
procedure GetNextTet32(tet_idx, idx_{0..3})
    next_tet_idx ← N_3^{tet_idx}
    for i ← 0, 2 do
        if idx_{exit_face_idx} = V_i^{tet_idx} then
            next_tet_idx ← N_i^{tet_idx}
        end if
    end for
    return next_tet_idx
end procedure
```

If the neighbor index points to a constrained face or tetrahedral mesh boundaries, we terminate the traversal. Otherwise, knowing the next tetrahedron, we discard p_i and idx_i by replacing its contents with the newly fetched point data p_3 and idx_3. We repeat this process until a geometry is intersected or the tetrahedral mesh boundaries are reached. In this method, no further modifications are necessary to ensure vertex ordering because the counterclockwise ordering is always preserved for points on the exit face.

Fetching a new vertex id requires three bitwise exclusive-or operations. The coordinate system transformation of the newly fetched point is six floating-point multiplications and four floating-point additions. We decide whether a face is an exit face by using four floating-point multiplications and two floating-point comparisons. Finally, we determine the next tetrahedron index using the appropriate method for the preferred structure.

Please confirm Appendix B for tetrahedron traversal algorithms for the alternative tetrahedral representations *Tet20* and *Tet16* and Appendix C for further traversal optimization using a special basis constructed from the ray direction for point projection. Additionally, Appendix D provides the implementation details about handling common ray-tracing operations.

5 Reordering tetrahedral mesh data

We reorder points and tetrahedra in memory to improve cache locality during ray traversal. For this purpose, we use a two-step method. In the first step, we detect if there are distinct regions in the tetrahedralization.

Each closed surface in the input geometry divides the space into two regions (outside and inside regions). These regions occur when a set of tetrahedra is completely enclosed by a set of constrained faces. Because the rays are traced until a constrained face is encountered, the tetrahedra from different regions are not visited in a single ray traversal, which is not the case for multi-hit traversal methods. Thus, we store the tetrahedra that belong to the same region close together in memory. Furthermore, we reorder points based on their positions and tetrahedra based on their center points. We map points to memory using a Hilbert curve (see Fig. 4). Hilbert curve is a space-filling curve that can be used to map spatial data from three dimensions to one dimension by preserving the locality. This means that primitives that are close to each other in 3D space are also close to each other in one dimension.

6 GPU Implementation

For the GPU implementation, we use the CUDA platform. Once we build the tetrahedral mesh-based acceleration structure, the tetrahedra and points data are copied to the GPU. We store the constrained face data on the host computer because it is not a part of the *hot* data, which is frequently accessed during traversal. Once initialization is complete, the steps to render a single frame are as follows.

1. We identify the *source tetrahedron* on the CPU.
2. We pass the batch of rays and the source tetrahedron to the global memory of the GPU.
3. CUDA kernels run for each ray, traversing the scene, and terminate when they hit the scene geometry.
4. We store the results of the intersection calculations in the global memory of the GPU and then passed them to the main memory. We use these results to perform shading and to generate additional rays.

Our method can be trivially implemented for the CUDA platform. However, this trivial implementation does not provide the best performance on the GPU in terms of computation speed. Thus, we perform the following optimizations to make our method run faster on the GPU.

1. We project ray origin to the 2D coordinate system beforehand. When projecting the newly fetched point, translation is performed on the 2D coordinate system instead of a 3D one. Thus, instead of using the origin in 3D, we use projected origin in 2D. This potentially results in fewer occupied registers on the GPU, resulting in better performance. We compute the projected origin, *po*, as follows:

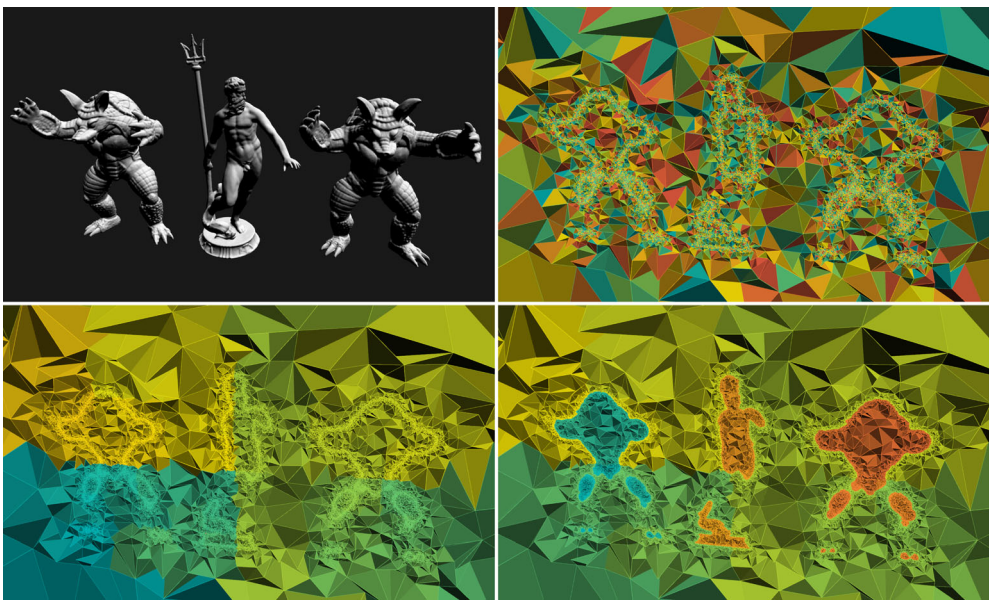


Fig. 4 Sorting tetrahedron data. Top left: The three-dimensional scene. Top right: Unsorted tetrahedron data. Bottom left: Tetrahedron data sorted using an Hilbert curve. Bottom right: Tetrahedron data sorted using an Hilbert curve and mesh regions. Memory positions are coded with different colors. Tetrahedra close in memory are represented with similar colors. Tetrahedron data are stored in a contiguous manner

$$po = (\mathbf{u} \cdot r_o, \mathbf{v} \cdot r_o), \tag{1}$$

where $(\mathbf{u}, \mathbf{v})$ is the 2D basis constructed from the ray. During traversal, we can project the new point to the 2D plane as follows:

$$p_3 = (\mathbf{u} \cdot v_{new} - po_x, \mathbf{v} \cdot v_{new} - po_y), \tag{2}$$

where p_3 is the projected point and v_{new} is the newly fetched point from the next tetrahedron.

2. We make use of CUDA textures when accessing tetrahedral mesh data. To optimize traversal in *Tet20* and *Tet16* structures, we use a single channel integer texture (32 bytes). To use it, the required elements are the xor field and one neighbor field for *Tet20*, the xor field and one or two neighbor fields for *Tet16*. We fetch and store these in local stack; potentially reducing the number of registers used.

7 Experimental results

We compare our approach to *k*-d tree, BVH, and the state-of-the-art tetrahedral mesh-based methods, namely the ScTP-based traversal (Lagae and Dutré 2008a) and the Plücker coordinate-based traversal (Maria et al. 2017b). We use the *k*-d tree and SAH-based BVH implementations, as described in Wald (2007); Gunther et al. (2007). We use the original implementation provided by Maria et al. (2017b) for the Plücker coordinate-based traversal.

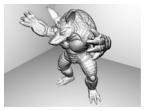
We use TetGen (Si 2015) to generate the tetrahedral mesh of the 3D scene. We perform experiments on a computer with six cores @3.2 GHz (Intel), 16 GB of main memory, and NVIDIA GTX 1060 with 6 GB of memory. On the CPU, we render the scenes at a resolution of 1920×1440 using multi-threading by subdividing the image into 16×16 tiles and assigning them to available threads. To make a fair comparison between our method and the other state-of-the-art approaches, we render the same scene many times and pick the best result for each method to avoid noisy measurements due to background processes.

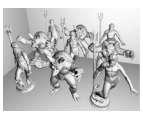
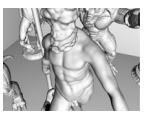
Tables 1 and 2 show the computational costs of the construction of acceleration structures and rendering times of different traversal methods. The test scenes in Table 2 cannot be tetrahedralized using TetGen. We tetrahedralized them using TetWild (Hu et al. 2018) and used the remeshed geometry as input. To test the adaptiveness of the structures in a challenging scene geometry, we include the versions of the scenes with bounding boxes composed of large triangles. Experiments show that our method performs better than the ScTP- (Lagae and Dutré 2008a) and Plücker coordinate-based traversal methods (Maria et al. 2017b) in all scenes. It performs better than the BVH-based traversal in seven of the fifteen scenes and better than the *k*-d tree-based traversal in six of the fifteen scenes. In the other scenes that BVH- and *k*-d tree-based traversal methods perform superior to our tetrahedral mesh-based traversal, the rendering times are mostly close to each other. While testing the state-of-the-art tetrahedral-mesh-based traversal methods of (Lagae and Dutré 2008a; Maria et al. 2017b), we sorted the tetrahedral meshes using space-filling curves for a fair comparison. Although the construction times of BVH and *k*-d tree are lower than that of the tetrahedral meshes, the tetrahedral mesh is constructed during preprocessing, and it does not affect the ray-tracing performance for the scenes that do not require the update of acceleration structures. The tetrahedral mesh does not need to be updated for dynamic scenes where the topology does not change. If the topological changes to a tetrahedralization are local, the tetrahedral mesh can be updated with efficient insertion and removal operations (Lagae and Dutré 2008a).

Table 3 shows rendering times of tetrahedral mesh-based methods for test scenes on the GPU. *Tet20* representation gives the best performance, which is around 15% faster than Maria's method while occupying much less memory (half of the memory required by Maria's method in the largest scene). *Tet16* representation requires even less memory but it is not as fast as *Tet20* (roughly the same performance as Maria's method) due to more memory and arithmetic operations needed to decode compressed neighbor data.

Table 4 shows the memory costs for different acceleration structures on different scenes. Our most compact structure, *Tet16*, can be stored using significantly less memory than the other tetrahedral-mesh-based alternatives, *Tet-mesh-ScTP* (Lagae and Dutré 2008a) and *Tet-mesh-80* (Maria et al. 2017b), which provides two benefits. First, accelerators for much larger scenes can fit the main memory or GPU global memory. Second, this small footprint provides much better performance by facilitating cache locality. Our smallest accelerator data are memory aligned (16 bytes per tetrahedron). Memory usage of the *k*-d tree tends to be affected considerably by the distribution of the primitives in the scene, thus resulting in a smaller or larger accelerator size in different scenes. On the other hand, memory used for a BVH structure is always

Table 1 Computational costs of acceleration structures and rendering times for traversal methods.

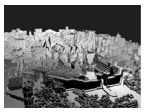
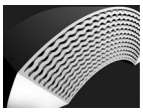
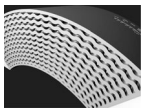
Scenes	Torus knots	Torus knots in a box	Armadillo	Armadillo in a box	Neptune
					
Scene statistics					
# of triangles	77,760	77,772	345,938	345,950	448,896
# of tetrahedra	270,036	270,351	1,027,749	1,021,965	1,240,582
Construction times (in seconds)					
Tet-mesh-ScTP	3.596	3.638	16.733	20.252	79.822
Tet-mesh-80	3.656	3.663	16.595	20.143	79.657
Tet-mesh-32	3.753	3.643	16.659	20.262	79.536
Tet-mesh-20	3.778	3.658	16.704	20.444	79.573
Tet-mesh-16	3.640	3.524	16.546	19.919	79.846
BVH	**0.078**	**0.079**	**0.391**	**0.396**	**0.474**
k-d tree	0.739	0.590	1.454	1.651	2.265
Rendering times (in milliseconds)					
Tet-mesh-ScTP	261.5	293.4	232.5	306.7	268.9
Tet-mesh-80	244.1	278.9	218.1	262.4	236.5
Tet-mesh-32	150.7	181.7	148.5	182.5	158.7
Tet-mesh-20	**125.8**	**142.3**	117.1	145.3	127.1
Tet-mesh-16	136.3	152.4	124.6	153.2	135.9
BVH	152.7	192.2	**78.1**	**126.1**	**78.7**
k-d tree	139.9	214.4	85.7	182.3	81.7

Scenes	Neptune in a box	Mix	Mix in a box	Mix close	Mix in a box close
					
Scene statistics					
# of triangles	448,908	2,505,992	2,506,004	2,505,992	2,506,004
# of tetrahedra	1,242,883	7,259,175	7,252,946	7,259,175	7,259,193
Construction times (in seconds)					
Tet-mesh-ScTP	155.478	124.208	170.216	124.840	485.950
Tet-mesh-80	156.381	125.081	169.116	125.068	482.908
Tet-mesh-32	153.788	124.000	169.502	123.183	487.291
Tet-mesh-20	155.580	124.087	169.890	124.015	483.827
Tet-mesh-16	154.644	124.493	170.175	123.401	484.516
BVH	**0.487**	**2.968**	**3.017**	**2.966**	**2.997**
k-d tree	2.471	13.889	14.668	13.846	16.624
Rendering times (in milliseconds)					
Tet-mesh-ScTP	279.6	402.6	430.0	449.5	455.0
Tet-mesh-80	261.0	355.7	384.7	411.2	419.6
Tet-mesh-32	176.1	247.2	268.5	265.5	269.9
Tet-mesh-20	137.0	196.3	211.1	205.6	**210.2**
Tet-mesh-16	152.0	223.9	241.4	237.4	240.3
BVH	**120.4**	144.6	**187.9**	224.9	253.8
k-d tree	162.6	**143.5**	214.8	**193.1**	213.6

We compare our *Tet-mesh-32*, *Tet-mesh-20*, and *Tet-mesh-16* structures and traversal methods with *Tet-mesh-ScTP* (Lagae and Dutré 2008a), *Tet-mesh-80* (Maria et al. 2017b), BVH (Pharr et al. 2016), and *k*-d tree (Pharr et al. 2016)

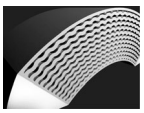
smaller than its tetrahedra-mesh-based counterparts. It should be noted that BVHs and tetrahedra-mesh-based accelerators exhibit orthogonal strengths, which can be combined for better results, as demonstrated in

Table 2 Computational costs of acceleration structures and rendering times for traversal methods (remeshed scenes).

Scenes	Rungholt far	Rungholt default	Rungholt close	Exhaust pipe left	Exhaust pipe right
					
Scene statistics					
# of triangles	3,580,928	3,580,928	3,580,928	6,244,678	6,244,678
# of tetrahedra	8,381,071	8,381,071	8,381,071	18,480,542	18,480,542
Construction times (in seconds)					
BVH	4.230	4.247	4.212	7.771	7.767
k-d tree	37.140	37.078	37.039	66.462	66.403
Rendering times (in milliseconds)					
Tet-mesh-ScTP	554.035	525.523	436.716	333.291	344.483
Tet-mesh-80	488.299	466.933	400.589	312.152	320.361
Tet-mesh-32	353.444	333.274	265.337	202.472	207.239
Tet-mesh-20	282.211	265.337	215.118	163.814	166.619
Tet-mesh-16	313.335	293.351	238.027	183.198	186.125
BVH	198.140	227.333	243.165	177.263	187.540
k-d tree	**119.589**	**127.948**	**126.434**	**114.358**	**121.114**

We compare our *Tet-mesh-32*, *Tet-mesh-20*, and *Tet-mesh-16* structures and traversal methods with *Tet-mesh-ScTP* (Lagae and Dutré 2008a), *Tet-mesh-80* (Maria et al. 2017b), BVH (Pharr et al. 2016), and *k*-d tree (Pharr et al. 2016)

Table 3 Rendering times of tetrahedral mesh-based acceleration structures on the GPU.

Scenes	Torus Knots	Armadillo	Neptune	Mix	Rungholt	Exhaust Pipe
						
Scene statistics						
# of triangles	77,760	345,938	448,896	2,505,992	3,580,928	6,244,678
# of tetrahedra	270,036	1,027,739	1,240,582	7,259,175	8,381,071	18,480,542
Kernel execution time (in milliseconds)						
Tet-mesh-ScTP	20.021	19.612	20.898	43.910	43.633	22.560
Tet-mesh-80	7.790	7.136	7.941	13.454	14.360	9.023
Tet-mesh-32	19.541	18.950	20.958	42.690	44.544	21.320
Tet-mesh-20	**6.156**	**5.803**	**6.529**	**11.322**	**12.172**	**6.931**
Tet-mesh-16	7.120	6.477	7.328	12.157	13.444	8.231

We compare our *Tet-mesh-32*, *Tet-mesh-20*, and *Tet-mesh-16* structures and traversal methods with *Tet-mesh-ScTP* (Lagae and Dutré 2008a) and *Tet-mesh-80* (Maria et al. 2017b)

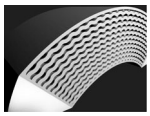
Aman et al. (2021). This approach allows dynamic scenes to be rendered by combining two structures: a top-level acceleration structure, a BVH, and a bottom-level acceleration structure, a tetrahedral mesh.

Figure 5 demonstrates the effect of the tetrahedral mesh sorting on rendering performance. Even though sorting is not vital for performance in small scenes, it significantly improves the rendering performance in large scenes. Please confirm Appendix E for the comparison of the performance of our tetrahedral mesh-based traversal with that of other acceleration structures based on the camera distance.

Although representations of Lagae and Dutre (2008a) and Maria et al. (2017b) and our method use similar mesh representations, the performance difference between them is because of the following reasons:

– *Memory operations per tetrahedron:* We only fetch one point per tetrahedron, thanks to the xor-based storage scheme. In *Tet-mesh-ScTP* and Plücker-based method, all four points are fetched from the

Table 4 Memory requirements of various acceleration structures.

Scenes	Torus knots	Armadillo	Neptune	Mix	Rungholt	Exhaust pipe
						
Scene statistics						
# of triangles	77,760	345,938	448,896	2,505,992	3,580,928	6,244,678
# of tetrahedra	270,036	1,027,739	1,240,582	7,259,175	8,381,071	18,480,542
Accelerator size (in megabytes)						
*k*d-tree	19.9	7.8	27.4	97.1	512.0	1417.8
BVH	4.7	20.4	10.6	150.0	175.6	380.0
Tet-mesh-ScTP	12.3	49.4	61.2	352.1	406.1	885.1
Tet-mesh-80	20.6	78.4	94.6	553.8	639.4	1410.0
Tet-mesh-32	12.3	49.4	61.2	352.1	406.1	885.1
Tet-mesh-20	9.2	37.7	47.0	269.0	310.2	673.6
Tet-mesh-16	8.2	33.7	42.2	241.3	278.2	603.1

We compare our proposed tetrahedra-mesh-based *Tet-mesh-32*, *Tet-mesh-20*, and *Tet-mesh-16* structures with the state-of-the-art tetrahedra-mesh-based acceleration structures, *Tet-mesh-ScTP* (Lagae and Dutré 2008a) and *Tet-mesh-80* (Maria et al. 2017b), and other types of acceleration structures, Bounding Volume Hierarchy (BVH), and *k*-d tree. Because *Tet-mesh-32* and *Tet-mesh-ScTP* use the same tetrahedral mesh representations, their memory requirements are the same

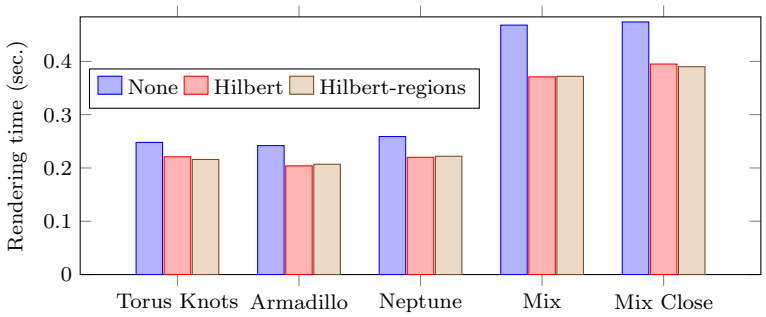


Fig. 5 Rendering times for unsorted and sorted tetrahedral mesh data

memory. Although three of them will be in the cache because three points are shared between tetrahedra, it still costs more than fetching only one point.

- *Compact storage:* Our method requires less memory than the approaches we compare. This speeds up the computations because the cache utilization is high. This also allows us to render larger scenes since more geometry can be fitted to the memory.
- *Arithmetic operations per tetrahedron:* Tet-mesh-ScTP relies on a scalar triple product, which accounts for 40 floating point operations on average to compute the terms. Similarly, the method proposed by Maria et al. (2017b) also works in 3D, thus resulting in more expensive computations. On the other hand, our transformed 2D coordinate system results in very few arithmetic operations (13 floating-point operations). Because we project points as soon as they are fetched from the memory, they occupy few registers. In *Tet-mesh-ScTP*, there may be a possible performance loss due to more register usage.
- *Determining the next tetrahedron:* In our method, we never take the previous tetrahedron into account as the next tetrahedron to visit (similar to Maria et al. (2017b)). However, *Tet-mesh-ScTP* takes all four neighbors into account by computing 3–6 scalar triple products to determine the next tetrahedron, which makes the computations more costly. Besides, this may reduce the effectiveness of the branch prediction as well because there are more candidate neighbors.

8 Conclusions and future research directions

We propose methods for fast tetrahedral mesh traversal for ray tracing. Specifically, we propose compact and memory-aligned tetrahedral mesh data structures. We use a space-filling curve to improve cache locality. We propose efficient traversal methods to improve ray-tracing performance and provide its GPU implementation. Experiments show that our approach can reduce rendering times substantially and perform better than other alternatives in different scenarios. There are two main limitations of using tetrahedral meshes as acceleration structures in ray-tracing complex 3D scenes.

- Tetrahedral mesh generation process is computationally costly and requires a significant amount of memory than the alternative methods.
- Our current implementation is not able to construct a tetrahedral mesh acceleration structure for scenes with intersecting geometry. We can overcome this limitation by a preprocessing step where mesh intersections are resolved so that the resulting geometry is a piecewise linear complex (Miller et al. 1996), as proposed in Lagae and Dutré (2008a).

Other areas for further research regarding contemporary ray-tracing concepts are as follows.

- *Non-triangular models:* The proposed acceleration structure does not support non-triangular models. Recent research by Hu et al. (2019) provide a way to build triangulations with curve constraints. The extension of this method to 3D with surface constraints can act as an accelerator, which could be a potentially interesting and challenging research direction.
- *Real-time rebuilds:* Although our approach allows real-time manipulation of the geometry by certain deformers (smooth, C1 continuous) naturally, it is not very easy to have real-time rebuilds on changing geometry, which is well-supported by the state-of-the-art BVHs.

We plan to experiment with the triangulations with curve constraints (Hu et al. 2019). The extension of this method to 3D would allow us to render parametric 3D surfaces directly using tetrahedralizations. Another potential use case is volume visualization. Even though the adaptation of our approach to direct volume rendering would result in a slower traversal (and possibly overlap with the approach employed by Marmitt et al. (2006)), there are still two potential improvements it can provide:

i) First, our compact structure would result in better cache utilization, reducing the computation time.
ii) Second, this structure would need little memory and enable visualization of large models that can fit into the memory. This is even more critical in GPU, where the memory is relatively limited.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s12650-022-00842-x.

Acknowledgements This research is supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. 117E881. We are grateful to Dr. Maxime Maria and his colleagues for providing us their implementation of the tetrahedral mesh traversal method.

References

Aman A, Demirci S, Güdükbay U, Wald I (2021) Multi-level tetrahedralization-based accelerator for ray-tracing animated scenes. Comput Anim Virtual World 32(3–4):e2024
Berk H, Aykanat C, Gudukbay U (2003) Direct volume rendering of unstructured grids. Comp & Graph 27:387–406
Bowyer A (1981) Computing dirichlet tessellations. Comput J 24(2):162–166
Duff T, Burgess J, Christensen P, Hery C, Kensler A, Liani M, Villemin R (2017) Building an orthonormal basis, revisited. J Comp Graph Tech 6(1):1–8
Edelsbrunner H, Shah NR (1992) Incremental topological flipping works for regular triangulations. In: Proc. Eighth Ann. Symp. Comp. Geom., ACM, New York, NY, USA, SCG '92, pp 43–52
Fellegara R, Floriani LD, Magillo P, Weiss K (2020) Tetrahedral trees: A family of hierarchical spatial indexes for tetrahedral meshes. ACM Trans Spat Algo Syst 6(4):23, 34 p
Fujimoto A, Tanaka T, Iwata K (1988) ARTS: accelerated ray-tracing system. In: Joy KI, Grant CW, Max NL, Hatfield L (eds) Tutorial: Computer Graphics. Image Synthesis, Computer Science Press Inc, New York, NY, USA, pp 148–159
Garrity MP (1990) Raytracing irregular volume data. In Proc. Eighth Joint Eurographics/IEEE VGTC Conf. Vis., ACM, New York, NY, USA, VVS '90, pp 35–40
Garth C, Joy KI (2010) Fast, memory-efficient cell location in unstructured grids for visualization. IEEE Trans Vis Comput Graph 16(6):1541–1550
Glassner AS (1984) Space subdivision for fast ray tracing. IEEE Comp Graph App 4(10):15–24

Goldsmith J, Salmon J (1987) Automatic creation of object hierarchies for ray tracing. IEEE Comp Graph App 7(5):14–20
Gunther J, Popov S, Seidel HP, Slusallek P (2007) Realtime ray tracing on GPU with BVH-based packet traversal. In Proc. IEEE Symp. Interactive Ray Tracing, IEEE Computer Society, Washington, DC, USA, RT '07, pp 113–118
Haines E, Greenberg D (1986) The light buffer: a shadow-testing accelerator. IEEE Comp Graph App 6(9):6–16
Havran V, Bittner J (2002) On improving kd tree for ray shooting. J WSCG 10:209–216
Hu Y, Zhou Q, Gao X, Jacobson A, Zorin D, Panozzo D (2018) Tetrahedral meshing in the wild. ACM Trans Graph 37(4):60
Hu Y, Schneider T, Gao X, Zhou Q, Jacobson A, Zorin D, Panozzo D (2019) TriWild: Robust triangulation with curve constraints. ACM Trans Graph 38(4):52
Hunt W, Mark W (2008) Adaptive acceleration structures in perspective space. In: Proc. IEEE Symp. Interactive Ray Tracing, RT '08, pp 11–17
Koyamada K (1992) Fast traverse of irregular volumes. In: Kunii TL (ed) Vis Comput. Springer Japan, Tokyo, pp 295–311
Lagae A, Dutré P (2008) Accelerating ray tracing using constrained tetrahedralizations. Comp Graph Forum 27(4):1303–1312
Lagae A, Dutré P (2008) Compact, fast and robust grids for ray tracing. Comput Graph Forum 27(4):1235–1244
MacDonald DJ, Booth KS (1990) Heuristics for ray tracing using space subdivision. Vis Comput 6(3):153–166
Maria M, Horna S, Aveneau L (2014) Topological space partition for fast ray tracing in architectural models. In: Proc. Int. Conf. Comp. Graph. Theory App., GRAPP '14, pp 1–11
Maria M, Horna S, Aveneau L (2017) Constrained convex space partition for ray tracing in architectural environments. Comput Graph Forum 36(1):288–300
Maria M, Horna S, Aveneau L (2017b) Efficient ray traversal of constrained Delaunay tetrahedralization. In: Proc. Int. Joint Conf. Comp. Vis., Imag. Comp. Graph. Theory Appl., VISIGRAPP '17, vol 1, pp 236–243
Marmitt G, Slusallek P (2006) Fast ray traversal of tetrahedral and hexahedral meshes for direct volume rendering. In: Proc. Eighth Joint Eurographics/IEEE VGTC Conf. Vis., Eurographics Assoc., Aire-la-Ville, Switzerland, EUROVIS '06, pp 235–242
Maximo A, Ribeiro S, Bentes C, Oliveira A, Farias R (2008) Memory efficient GPU-based ray casting for unstructured volume rendering. In: Proc. Fifth Eurographics / IEEE VGTC Symp. Point-Based Graphics, Eurographics Assoc., Goslar, DEU, SPBG'08, pp 155–162
Mebarki A (2018) XOR-based compact triangulations. Comp & Inform 37:367–384
Miller GL, Talmor D, Teng SH, Walkington N, Wang H (1996) Control volume meshes using sphere packing: Generation, refinement and coarsening. In: Proc. 5th Int. Meshing Roundtable, pp 47–61
Pharr M, Jakob W, Humphreys G (2016) Physically based rendering: from theory to implementation, 3rd edn. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA
Platis N, Theoharis T (2003) Fast ray-tetrahedron intersection using Plücker coordinates. J Graph Tool 8(4):37–48
Ribeiro S, Maximo A, Bentes C, Oliveira A, Farias R (2007) Memory-aware and efficient ray-casting algorithm. In: Proc. XX Brazilian Symp. Comp. Graph. Img. Process., SIBGRAPI '07, pp 147–154
Sahistan A, Demirci S, Morrical N, Zellmann S, Aman A, Wald I, Güdükbay U (2021) Ray-traced shell traversal of tetrahedral meshes for direct volume visualization. In: Proc. IEEE Vis. Conf.-Short Papers, VIS '21, pp 91–95
Shewchuk JR (1996) Adaptive precision floating-point arithmetic and fast robust geometric predicates. Disc & Comp Geom 18:305–363
Si H (2015) TetGen, a Delaunay-based quality tetrahedral mesh generator. ACM Trans Math Soft 41(2):11, 36 p
Silva CT, Mitchell JSB (1997) The lazy sweep ray casting algorithm for rendering irregular grids. IEEE Trans Vis Comp Graph 3(2):142–157
Silva CT, Mitchell JSB, Kaufman AE (1996) Fast rendering of irregular grids, Proc. Symp. Vol. Vis., VIS '96, 1996, pp. 15–22
Sinha P (2004) A memory-efficient doubly linked list. Linux J, Available at https://www.linuxjournal.com/article/6828. Accessed 5 May 2022
Stich M, Friedrich H, Dietrich A (2009) Spatial Splits in Bounding Volume Hierarchies. In: Proc. Conf. High Perf. Graph., ACM, New York, NY, USA, HPG '09, pp 7–13
Wald I (2007) On fast construction of SAH-based bounding volume hierarchies. In: Proc. IEEE Symp. Interactive Ray Tracing, IEEE Computer Society, Washington, DC, USA, RT '07, pp 33–40
Wald I, Havran V (2006) On building fast kd-trees for ray tracing, and on doing that in O(N log N). In: Proc. IEEE Symp. Interactive Ray Tracing, IEEE Computer Society, Washington, DC, USA, RT '06, pp 61–69
Watson DF (1981) Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. Comput J 24(2):167–172
Wodniok D, Goesele M (2017) Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees. Comp & Graph 62:41–52