Ray-traced Shell Traversal of Tetrahedral Meshes for Direct Volume Visualization

Alper Sahistan* Bilkent University
Serkan Demirci Bilkent University
Nathan Morrical University of Utah
Stefan Zellmann University of Cologne
Aytek Aman Bilkent University
Ingo Wald NVIDIA Corporation
Uğur Güdükbay Bilkent University

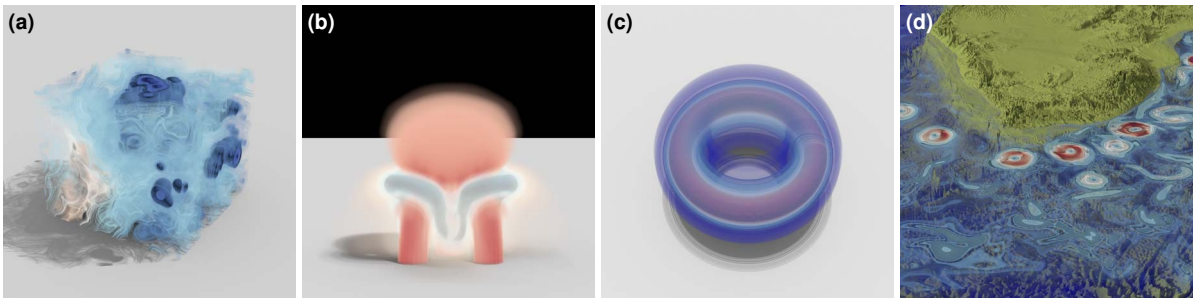

Figure 1: Renderings with secondary effects obtained with the proposed approach on an NVIDIA RTX 8000 GPU: (a) *Plasma64* dataset rendered at 46.6 frames per second (fps). (b) *Jets* dataset rendered at 48.5 fps. (c) *Fusion* dataset rendered at 13.9 fps. (d) *Agulhas* dataset rendered at 5.6 fps.

Abstract

A well-known method for rendering unstructured volumetric data is tetrahedral marching (*tet marching*), where rays are marched through a series of tetrahedral elements. However, existing tet marching techniques do not easily generalize to rays with arbitrary origin and direction required for advanced shading effects or non-convex meshes. Additionally, the memory footprint of these methods may exceed GPU memory limits. Interactive performance and high image quality are opposing goals. Our approach significantly lowers the burden to render unstructured datasets with high image fidelity while maintaining real-time and interactive performance even for large datasets. To this end, we leverage hardware-accelerated ray tracing to find entry and exit faces for a given ray into a volume and utilize a compact mesh representation to enable the efficient marching of arbitrary rays, thus allowing for advanced shading effects that ultimately yields more convincing and grounded images. Our approach is also robust, supporting both convex and non-convex unstructured meshes. We show that our method achieves interactive rates even with moderately-sized datasets while secondary effects are applied.

Index Terms: Human-centered computing—Visualization—Visualization application domains—Scientific visualization; Computing methodologies—Computer Graphics—Rendering—Ray Tracing

1 Introduction

One of the most common techniques to render scientific datasets is direct volume rendering (DVR). Direct volume rendering is usually performed using ray traversal, which allows for evaluating advanced shading effects such as ambient occlusion or single or multiple scattering. The long-established methods for *unstructured meshes*; however, often use a rasterization framework to initiate and perform raymarching and thus do not allow for simple integration of these effects. A well-known method for unstructured meshes is ray-marching via connectivity information, where each ray samples and integrates the opacity and color information for each cell they pass through. DVR techniques that can render unstructured volumes come with their own challenges. We aim to tackle these challenges while avoiding the shortcomings of raster-based approaches.

Unstructured volume traversal via face connectivity information is often met with challenging limitations like high memory usage and robustness issues. These issues usually hinder performance or exceed the memory limitations of modern GPUs. We address these challenges for non-raster frameworks using compact memory layouts and robust ray-tetrahedron intersection methods.

To further improve visualization quality and perception, we trace arbitrary rays to apply visual effects such as ambient occlusion, shadows, shading, and border contours (see Fig. 1). These effects usually require tracing secondary rays, cast after primary rays interact with the scene geometry. Unlike camera rays, these secondary rays are arbitrary; achieving these effects in raster-based pipelines is challenging [10]. In addition, existing solutions usually include extra raster passes, hindering interactive performance.

We propose a novel approach to ray-marching unstructured meshes that combine three main improvements:

- a compact, cache- and GPU-friendly memory layout that facilitates fast ray-tetrahedra intersection and efficient tetrahedra-to-tetrahedra traversal,
- the ability to handle arbitrary, non-common-origin rays as required for secondary effects like ray-traced reflections, shadows, or ambient occlusion, and
- the ability to efficiently handle convex and non-convex datasets, datasets with holes, curves, and discontinuities.

2 Related Work

Several research works have recently proposed to exploit RTX hardware for DVR [14, 20, 23, 24]. Wald et al. [23] use RTX BVHs for point location in unstructured tetrahedral meshes, at the cost of considerable memory consumption. The authors used these point

*e-mail:alper.sahistan@bilkent.edu.tr

978-1-6654-3335-8/21/$31.00 ©2021 IEEE
DOI 10.1109/VIS49827.2021.00026

location kernels to accelerate a sampling-based ray marcher for tet meshes. This work was later extended by Morrical et al. [15] to support other cell types than tetrahedra. In contrast to these approaches, we use BVHs for just the exterior (shell) faces of tetrahedral volumes. Our work is orthogonal to that by Muigg et al. [16], which is tailored to the rasterization pipeline. They first subdivide the volume into bricks using a kd-tree and then render these in front-to-back order. Then they perform a depth peeling step where each brick's entry and exit faces are determined. In contrast to theirs, our method allows us to trace secondary rays whose origin and direction are different from those of viewing rays.

When raymarching through tetrahedral volumes, we need to determine the next tetrahedron at every step. Several methods have been proposed to accomplish that [1,7–9]. Our work is based on that by Aman et al. [1] who proposed highly optimized memory layouts to improve marching performance and reduce memory consumption. Their method is based on projecting tetrahedra vertices into a 2-D ray-centric coordinate system to reduce instruction count. We extend their raymarching algorithm to support DVR.

3 Method Overview

In order to support high-quality DVR, we need to be able to a) find entry faces and their associated tetrahedra, and b) to efficiently *march* from one tetrahedron to the next, a method we call *tetrahedra marching*, or *tet marching*. The process of finding entry and exit faces on what we call the *shell* (the triangle mesh induced by the generally non-convex hull of the tet mesh) we accelerate using an OptiX bounding volume hierarchy (BVH) [11] (the *shell-BVH*). Traditional raster-based approaches like the one by Muigg et al. [16] use multiple passes for finding entry faces. Here, each pixel is assigned the tetrahedron ID where marching starts. While powerful, these approaches do not easily extend to non-convex meshes and only allow for common-origin projection. This work demonstrates how we can address these shortcomings by using a dedicated ray-tracing framework like OptiX. These adjustments allow for arbitrary ray / tet-mesh intersections and thus for high-quality ray-traced shading effects that, to our knowledge, have not been demonstrated in a tetrahedra marching framework before.

Efficient tetrahedra marching relies on the principle of ray-connectivity between elements. In this work, all primitives are tetrahedra, which are occasionally derived by tetrahedralizing higher-dimensional elements. Ray-connectivity implies that each tetrahedron neighbors another tetrahedron if they share one of their four faces, which makes up a continuous path of tetrahedra on a ray segment. If a face does not connect two tetrahedra, this makes that face a shell-face by definition. The tetrahedra marching method operates between these shared faces where rays go face to face until they reach an opaque region or a termination condition is met.

Our marchers are optimized to reduce memory accesses, cache misses, and arithmetic complexity. We exploit the fact that each tetrahedron shares three vertices with a neighbor to minimize memory access. Furthermore, we use a specialized 2-D projection to reduce arithmetic complexity. Furthermore, our memory scheme stores, sorts, and compresses neighborhood and vertex information.

4 Shell-to-Shell Traversal

The shell-BVH is just an ordinary BVH of triangles that we realize using the OptiX framework to leverage hardware acceleration for triangle geometry. The OptiX API requires us to specify the shell triangles as one list of triangle vertices and another list of triangle indices. Memory-wise those lists come on top of the already stored tetrahedron data structures as proposed in Sect. 5.1. For the indices, we use `int4`'s so that we can store, in addition to the triangle indices, the index of the tetrahedron that this triangle is associated with and that we use as an entry point for tet marching (see Sect. 5)

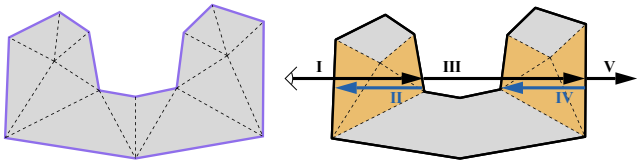

Figure 2: Shell-to-shell traversal in 2-D. Left: a non-convex tetrahedral volume with the shells in purple. Right: Example path where I) is the viewing ray. II) is a ray (blue) we cast *backward* to find the entry shell-face of the current volume segment. III) takes us to the next segment after tet marching the first segment completed. Finally, IV) is again cast to find the entry face and V) is cast to find the exit face for the second volume segment.

To initiate tet marching, we first need to find an entry face on the shell. This is simple if the entry face is in front of the ray origin but more involved when we start marching inside the volume and the entry face is *behind* the ray origin. We, therefore, generally first find the *exit* face by tracing a ray against the shell BVH, but with front face culling activated. We then trace another ray in the opposite direction, starting at the point of intersection with the exit face, to find the entry face. We march through the segment—potentially using early ray termination if the ray is used to compute radiance and then exit altogether. We then extend the ray and repeat that until all segments were processed. The process is illustrated in Fig. 2 and in Alg. 1, which we realized using an OptiX RayGen program [12].

Algorithm 1 Shell-to-shell traversal.

```
1:  procedure SHELL2SHELLTRAVERSAL(ray, shells, tetMesh)
2:      C_dst ← (0,0,0)                                   ▷ final color
3:      α ← 0                                             ▷ opacity
4:      while α < 1 do
5:          payload ← traceRay(ray, shells, CULLFRONT)
6:          if payload.hit then
7:              bRay.origin ← payload.hitPoint            ▷ backwards ray
8:              bRay.direction ← −ray.direction
9:              bPayload ← traceRay(bRay, shells, CULLFRONT)
10:             v_id_{0,...,2} ← bPayload.face_{0,...,2}  ▷ face vert. ids
11:             Id_tet ← Payload.face_3                   ▷ entry tet. id
12:             C_vol, α_vol ←
                    marchVolume(ray, v_id_{0,...,2}, Id_tet, tetMesh)
13:             C_dst += C_vol × α_vol × (1 − α)
14:             α += (1 − α) × α_vol
15:             ray.origin ← bRay.origin
16:         else                                          ▷ Nothing left to hit
17:             C_dst += C_miss × (1 − α)
18:             break
```

5 Efficient Tetrahedra Marching

We propose volume rendering variants of three memory layouts, *Tet32*, *Tet20*, and *Tet16*, with their marching algorithms, originally proposed by [1]. In the proposed structures, we store vertices in a separate list along with scalars. These layouts aim to reduce the memory footprint of the method while sustaining fast traversal times. We modify their traversal algorithm to approximate the volume rendering equation [5] and to work as a RayGen [12]. Furthermore, we extend the algorithm to trace arbitrary rays inside the tetrahedral mesh, allowing us to achieve advanced effects.

5.1 Memory Layouts

The tetrahedron representations that we use [1] have an exclusive-or-sum (xor-sum) field in common. This field, called `vx` in Fig. 3, allows us to reduce memory requirements, similar to [18] and [13]. We calculate the xor-sum as $vx^i = v_0^i \oplus v_1^i \oplus v_2^i \oplus v_3^i$. Since $a \oplus a = 0$ we can obtain a vertex index from vx^i given we know the other three.

In addition to the xor field, the structures contain up to three vertex and four neighbor indices, denoted as $\mathtt{v}_i$ and $\mathtt{n}_j$, respectively.

```
struct Tet32{
    int3 v;
    int  vx;
    int4 n; };
```

```
struct Tet20{
    int  vx;
    int4 n; };
```

```
struct Tet16{
    int  vx;
    int3 nx; };
```

Figure 3: Tetrahedra memory layouts: each integer is four bytes. *Tet32*, *Tet20*, and *Tet16* occupy 32, 20, and 16 bytes, respectively.

Tet32 layout stores all connectivity information, i.e., we use four neighbors of the tetrahedron along with three vertices and the xor field to obtain the 4^{th} vertex. *Tet20* layout stores all connectivity information for a tetrahedron, i.e., four of its neighbors along with the xor field. Because neighboring tetrahedra share three vertices during tetrahedra marching, we obtain the unshared vertex of the adjacent tetrahedron using the xor field. On top of that, in *Tet16* representation, instead of storing the indices of the neighboring tetrahedra, we keep values that can reconstruct indices along with the xor field. We pre-compute these fields as $nx^i_j = n^i_j \oplus n^i_3$ where $j \in \{0,1,2\}$ and i is tetrahedron index.

5.2 Marching Algorithms

Aman et al. [1] assume that the camera is inside the volume; they use a "source tetrahedron" to start traversal. While this assumption is valid in their case, this is, in general, only true for DVR if the camera is inside the mesh. Instead, we utilize the tetrahedron index information explained in Sect. 4 to compute entry and exit faces, thus initializing marching. Due to our compaction, without a tetrahedron index or a source tetrahedron, tet marching cannot be initiated.

Another critical component is the intersection tests that determine the tetrahedron to traverse next. We adopt the *point projection on a specialized basis* method proposed by Aman et al. [1] where points are projected to a 2-D ray centric space whose origin coincides with the ray origin and whose z-axis is the ray's direction vector. We start marching by obtaining the missing vertex index using the xor ($\oplus$) field of the current tetrahedron. At each step, we get new vertices using the xor field and perform front-to-back color compositing. The accumulated opacity is obtained through Beer's Law [5, 6, 19]. We use early ray termination with a threshold of 98%.

Each memory layout given in Sect. 5.1 comes with its marching algorithm, which shares the same skeleton. The differences come from unwrapping the applied compaction schemes. Alg. 2 summarizes the marching procedure for volume rendering. Algs. 1 and 2 can be generalized by changing the return types and sampling function.

Algorithm 2 Tetrahedra marching. *v_id* is the vertex index list for the current tetrahedron, *index* is the tetrahedron index and *tetMesh* is a list of tetrahedra, represented with one of *Tet32*, *Tet20*, or *Tet16*.

1: **procedure** MARCHVOLUME(*ray*, $v\_id_{0,\ldots,2}$, *index*, *tetMesh*)
2: $C_{vol}, \alpha \leftarrow (0,0,0),\ 0$
3: $v\_id_3 \leftarrow tetMesh_{index}.VX \oplus v\_id_0 \oplus v\_id_1 \oplus v\_id_2$ ▷ 3rd v_id
4: $V'_{0,\ldots,3} \leftarrow projectToBasis(Vert_{v\_id0,\ldots,3})$ ▷ points in ray-space
5: $exitFaceId \leftarrow GetExitFace(V'_{0,\ldots,3})$
6: **while** $index \neq -1$ AND $\alpha < 1$ **do**
7: $C_{vol}, \alpha \leftarrow sample(V'_{0,\ldots,3}, \ldots v\_id_{0,\ldots,3})$
8: $index \leftarrow marchToNextTet(V'_{0,\ldots,3}, exitFaceId, tetMesh_{index})$
9: $v\_id_{exitFaceId} \leftarrow v\_id_3$
10: $V'_{exitFaceId} \leftarrow V'_3$
11: $v\_id_3 \leftarrow tetMesh_{index}.VX \oplus v\_id_0$ ' $\oplus\, v\_id_1 \oplus v\_id_2$)
12: $V'_3 \leftarrow projectToBasis(Vert_{v\_id_3})$ ▷ project new point
13: $exitFaceId \leftarrow GetExitFace(V'_{0,\ldots,3})$
14: **return** C_{vol}, α

In order to connect to the next tetrahedron with *Tet32*, we determine which neighbor is behind the exit face based on the rank of the exit face's index in the current tetrahedron's vertex index list. For instance, let the exit face index be 42 and let the current tetrahedron's vertex indices be $\{32, 20, 42, 10\}$. Then, the algorithm will pick the third neighbor index because 42 is the third in the list.

The Tet20 representation does not explicitly store vertex indices. Initially, we sort each neighbor index using its corresponding vertex index, i.e., a vertex that does not share an edge with that tetrahedron. During marching, to get the next tetrahedron, we pick n^{th} neighbor stored at the current tetrahedron where n is the sorted order of the last vertex index. For instance, let the current tetrahedron's vertex indices be $\{20, 10, 42, 32\}$. Then algorithm picks the rank of the last vertex index as the next neighbor to be visited. When sorted, 32 falls into third place; hence, the third neighbor's index is picked.

Marching to the next tetrahedron for the *Tet16* representation is a bit more complicated since it inherits *Tet20*'s compaction steps and also reduces neighborhood information. During traversal, unlike *Tet20*, we also keep the index of the previous tetrahedron. In this way, we can extract the next tetrahedron index from the `nx` fields solving $n_j = nx_j \oplus n_3$ where j is the exit face index.

6 Increasing Rendering Quality

We describe how to implement several effects to achieve high-quality shading by tracing secondary rays. Using our marching algorithms that support traversal from arbitrary locations and in random directions, secondary rays naturally integrate with our framework.

6.1 Gradient Calculation

Gradients are commonly used as surface normals for local shading. As our method supports traversal starting at arbitrary origins, central-difference gradients [3, 5, 21] can be computed by marching six rays in orthogonal directions starting at the sample position, giving us accurate, high-quality gradients even if the gradient sample positions Δx fall outside the tetrahedron that the current sample is inside. As this exact method is relatively costly, we restrict that to high-density regions (above 80%). Our marchers can, however, only *stop* at tet faces and not at arbitrary positions, whereas the gradient sample positions will generally fall somewhere in-between. While we could imagine using a more exact scheme or an interpolation method as proposed by Shirley and Tuchman [17], we found it acceptable in practice to evaluate the gradients at the position $\Delta x'$ where the marcher stops and then divides the sample value by $\Delta x'$. Gradient shading with the Phong model is demonstrated in Fig. 4 and Fig. 5.

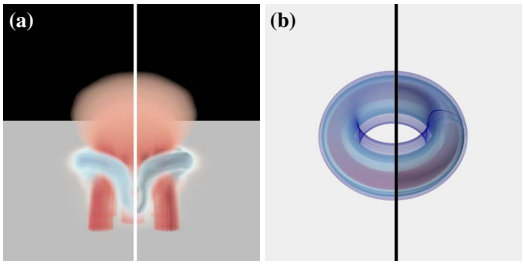

Figure 4: Gradient-shaded depth cues: (a) Jets dataset with shading off (left) and on (right) (113.3 fps vs. 70.45 fps). (b) Fusion dataset with shading off (left) and on (right) (87.3 fps vs. 12.2 fps).

6.2 Volumetric Shadows

Another way to improve depth perception is by rendering shadows that involve tracing arbitrary rays, thus allowing volumes to cast shadows on surfaces. To calculate the radiance that reaches a certain point, we cast a shadow ray from that point using shell-BVH. If the shadow ray hits a volume shell, we start tetrahedra marching to

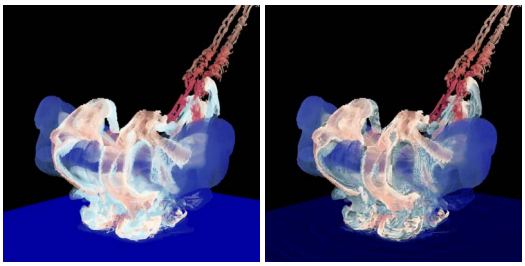

Figure 5: The Impact dataset. Left: with emission and absorption. Right: with gradients.

accumulate transmittance. Termination conditions and traversal are the same as Alg. 2.

Marching needs to be initiated from a shell-face. If a volume and surface mesh intersect, we cannot simply start from the tetrahedron containing the incident point. Because we are only interested in the transmittance and not in the radiance emitted from the volume, the order of tetrahedra on the shadow ray's path does not matter. Instead of marching in the direction of the shadow ray, we go backward, starting from the closest back-facing shell. If the light source lies inside the volume, we start accumulating transmittance after passing through the light's position. Fig. 6 (c) displays this effect.

6.3 Ambient Occlusion

Ambient occlusion (AO) can help even more with depth perception and overall rendering quality. We use the standard ray traced AO method, for example, proposed in [2,4]. Tracing the required shadow rays is technically very similar to tracing rays towards point light sources located at a distance of r. We compute AO by averaging N hemisphere samples, the effect of which can be seen in Fig. 6 (b).

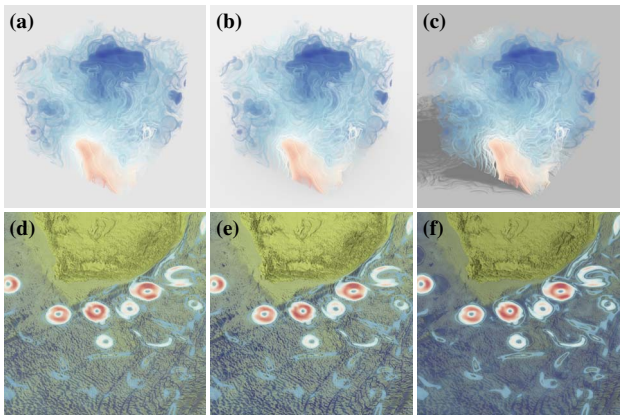

Figure 6: Shadows and ambient occlusion: (a) Plasma64 dataset rendered with emission and absorption at 141.5 fps, (b) with AO at 32.3 fps, and (c) with shadows at 104.1 fps. (d) Agulhas dataset rendered with emission and absorption at 42.4 fps, (e) with AO at 4.5 fps, and (f) with shadows at 30.9 fps.

7 Implementation and Evaluation

Our implementation is based on OptiX 7 [11] and the *OptiX Wrappers Library* (OWL) [22], which allows us to make use of NVIDIA's hardware ray tracing extensions. We use a ray generation program to initiate traversal and use `optixTrace` to trace rays against the shell's triangle mesh. Our tetrahedra structures (see Sect. 5.1) reside in arrays in GPU memory.

Experiments are performed on a workstation with an Nvidia Quadro RTX8000 GPU and Ubuntu 18.04. We evaluate our approach on various tetrahedral volume meshes with varying memory sizes, some of them coming with surface meshes always at 1024^2 resolution. We test each scene with all of our memory layouts with their respective marchers (see Sect. 5.1 and Sect. 5.2).

Table 1: Rendering times and memory usage for various scenes, with emission+absorption (E+A) vs. with all effects (AO, volumetric shadows, gradient shading). GPU memory measured with `nvidia-smi` is reported for the Tet20 memory layout (for the smaller data sets those numbers include a slight bias for constant overhead from the desktop environment). For Impact we deactivate shadows and AO for lack of a surface mesh.

	Rendering times (fps)			Memory usage		
Scene	Tet32	Tet20	Tet16	No. Tets	No. Shells	Mem. (MB)
Jets w/ E+A	38.92	116.84	101.51	1M	14K	541
Jets w/ effects	12.87	48.54	40.77			
Plasma64 w/ E+A	124.82	271.11	255.09	1.3M	49K	551
Plasma64 w/ effects	19.72	46.60	40.52			
Fusion w/ E+A	38.94	109.74	96.19	2.9M	89K	593
Fusion w/ effects	5.08	13.93	11.54			
Agulhas w/ E+A	7.20	22.10	22.54	201M	1.1M	5,313
Agulhas w/ effects	1.54	4.40	4.24			
Impact	13.64	25.54	24.95	366M	28M	10,453
Impact w/ effects	4.00	5.68	4.56			

8 Summary and Discussion

We propose shell traversal-based tetrahedra marching algorithms for direct volume rendering. Our method offers high cache coherence while tracing arbitrary rays, generating secondary effects such as gradient shading, shadows, and ambient occlusion. It also benefits from NVIDIA's ray-tracing cores to achieve hardware acceleration.

Table 1 provides the computational cost of our implementation for the tested scenes with various secondary effects using different tetrahedra representations. In all cases, the *Tet20* and *Tet16* representations outperform the *Tet32* layout. When we compare the *Tet20* and *Tet16* representations, there is no significant difference in performance. Using *Tet20* or *Tet16* has some practical advantages, and we can select one depending on the memory constraints.

We also evaluate ray tracing overhead (see Table 1). We observe that the memory overhead of OptiX scales reasonably well with the number of shell faces. We see that the shell-face count is considerably less than the total vertex count for the given scenes. Besides, the shell traversal cost constitutes $\approx 25\%$ of the total rendering time for our test ($\approx 75\%$ is tet-marching plus effects). We show that, unlike raster-based methods, applying any secondary effect that requires tracing arbitrary rays is cheap and easy with our method.

Although our approach reduces memory consumption while allowing efficient DVR, it has some limitations. Our method only works with pure tetrahedral-meshes. Many large volumetric datasets fail to comply with this constraint. Other types of meshes can be tetrahedralized, but the tetrahedralization of different primitives increases the memory cost. Some large datasets cannot be fit into the Video Random Access Memory on the GPU when tetrahedralized. Additionally, our marching procedure tends to go to the wrong tetrahedron and circle around when it encounters degenerate faces or tetrahedra. After a few iterations, we break these loops; they do not produce any noticeable artifact to our observation.

Acknowledgments

This research is supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under No. 117E881. The Agulhas dataset is courtesy of Dr. Niklas Röber (DKRZ); the Deep Water Asteroid Impact is courtesy of John Patchett and Galen Gisler of LANL. Plasma64 dataset is courtesy of AIM@SHAPE. Jets and Fusion datasets are courtesy of the SCI Institute of the University of Utah. Hardware for development and testing was graciously provided by NVIDIA Corp.

REFERENCES

[1] A. Aman, S. Demirci, and U. Güdükbay. Compact tetrahedralization-based acceleration structure for ray tracing. *arxiv preprint*, arXiv:2103.02309, 2021.

[2] M. Ament, F. Sadlo, C. Dachsbacher, and D. Weiskopf. Low-Pass Filtered Volumetric Shadows. *IEEE Tran. Vis. Comp. Graph.*, 20(12):2437–2446, 2014.

[3] C. Brownlee and D. Demarle. Fast volumetric gradient shading approximations for scientific ray tracing. In A. Marrs, P. Shirley, and I. Wald, eds., *Ray Tracing Gems II*. Apress Open, 2021.

[4] J. Díaz, P.-P. Vázquez, I. Navazo, and F. Duguet. Real-time ambient occlusion and halos with summed area tables. *Comp. & Graph.*, 34(4):337–350, 2010.

[5] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel. *Real-Time Volume Graphics*. A. K. Peters, Ltd., USA, 2006.

[6] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. In *ACM Comp. Graph.*, SIGGRAPH '84, p. 165–174. ACM, New York, NY, USA, 1984.

[7] A. Lagae and P. Dutre. Accelerating ray tracing using constrained tetrahedralizations. In *Proc. IEEE/EG Symp. Int. Ray Tracing*, p. 184, 2008.

[8] M. Maria, S. Horna, and L. Aveneau. Efficient ray traversal of constrained Delaunay tetrahedralization. In *Proc. Int. J. Conf. Comp. Vis., Imag. Comp. Graph. Theo. App.*, VISIGRAPP '17, pp. 236–243, 2017.

[9] G. Marmitt and P. Slusallek. Fast ray traversal of tetrahedral and hexahedral meshes for direct volume rendering. In *EUROVIS - Eurographics /IEEE VGTC Symp. on Vis.* Eurographics, 2006.

[10] N. Max. Optical models for direct volume rendering. *IEEE Trans. Vis. Comp. Graph.*, 1(2):99–108, 1995.

[11] NVIDIA Corp. NVIDIA OptiX Ray Tracing Engine. Available at https://developer.nvidia.com/optix, Accessed at 3 May 2021.

[12] NVIDIA Corporation. The Ray-Generation Program. Available at https://developer.nvidia.com/blog/how-to-get-started-with-optix-7/, Accessed at 3 May 2021.

[13] A. Mebarki. XOR-based compact triangulations. *Computing and Informatics*, 37:367–384, 2018.

[14] N. Morrical, W. Usher, I. Wald, and V. Pascucci. Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In *Proc. IEEE Vis.*, pp. 256–260, 2019.

[15] N. Morrical, I. Wald, W. Usher, and V. Pascucci. Accelerating unstructured mesh point location with RT cores. *IEEE Trans. Vis. Comp. Graph.*, pp. 1–1, 2020.

[16] P. Muigg, M. Hadwiger, H. Doleisch, and E. Groller. Interactive volume visualization of general polyhedral grids. *IEEE Trans. Vis. Comp. Graph.*, 17(12):2115–2124, 2011.

[17] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *SIGGRAPH Comput. Graph.*, 24(5):63–70, Nov. 1990. doi: 10.1145/99308.99322

[18] P. Sinha. A memory-efficient doubly linked list. *Linux Journal*, November 2004. Available at https://www.linuxjournal.com/article/6828, Accessed 3 May 2021.

[19] C. M. Stein, B. G. Becker, and N. L. Max. Sorting and hardware assisted rendering for volume visualization. In *Proc. IEEE Symp. Vol. Vis.*, VVS '94, 1994.

[20] D. Ströter, J. Mueller-Roemer, A. Stork, and D. Fellner. OLBVH: Octree Linear Bounding Volume Hierarchy for Volumetric Meshes. *The Vis. Comp.*, 36(10-12):2327–2340, 2020.

[21] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil. OSPRay - a CPU ray tracing framework for scientific visualization. *IEEE Trans. Vis. Comp. Graph.*, 23(1):931–940, 2017.

[22] I. Wald, N. Morrical, and E. Haines. OWL – The Optix 7 Wrapper Library, 2020. Available at https://github.com/owl-project/owl.

[23] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci. RTX beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location. In *Proc. High-Perf. Graph. - Short Papers*. Eurographics, 2019.

[24] I. Wald, S. Zellmann, W. Usher, N. Morrical, U. Lang, and V. Pascucci. Ray tracing structured AMR data using ExaBricks. *IEEE Trans. Vis. Comp. Graph.*, 27(2):625–634, 2021.