Interval Shading: using Mesh Shaders to generate shading intervals for volume rendering

Thibault Tricard

Interval Shading: using Mesh Shaders to generate shading intervals for volume rendering

THIBAULT TRICARD, Maverick, INPG, LJK, Inria, France

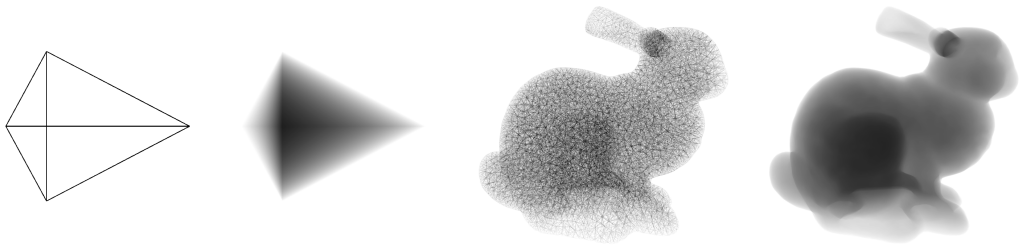

Fig. 1. Example of use of our method for volume rendering. From left to right: wireframe of a tetrahedron, the difference in depth between the front and back faces of the tetrahedron, wireframe of the Stanford bunny tetrahedrized (61 042 tetrahedrons), the sum of the depth difference between front and back faces of the tetrahedrons of the bunny rendered in a single pass in 3.30 milliseconds.

We propose to use tetrahedrons as primitives for volume rendering and a pipeline to rasterize them. Our work relies on the recently introduced mesh shaders to encode each tetrahedron such that the rasterizer computes the depths of the front and back faces at the same time when interpolating vertices attributes. Then, the fragment shader receives the two depths and can compute its shading in the interval. Our method is simple to implement, efficient, and opens new possibilities for the rasterization pipeline.

1 INTRODUCTION

In computer graphics volume rendering is an essential tool for representing complex scenes. Volume rendering can be used to represent complex phenomena (clouds [Bouthors et al. 2008], particle effects [Cha et al. 2009], etc.) for the creation of VFX in movies or video games. It can also be used for the visualization of complex mechanical systems in CAD [Huang and Carter 2005], for scientific visualization [Anderson et al. 2007], and for representing transparent materials [Everitt 2001].

Nowadays graphic cards are optimized to render surfaces and support the rendering of points, edges, and triangles. However, despite the popularity of volume rendering, there is no hardware acceleration for the rasterization of 3D primitives. As a result, volume rendering methods do not take full advantage of the graphic card and the vast majority of these algorithms are executed in fragment shaders. This can be explained by the similarity of the rasterization for 0, 1, and 2D primitives as solving their on-screen projection create a single solution per pixel: a fragment with a depth, interpolated attributes, etc. In comparison, solving the on-screen projection of a 3D primitive creates multiple solutions for each pixel that have to be solved simultaneously, stored contiguously, and sorted according to their depth. Doing so naively would create a bottleneck in the rendering pipeline by introducing synchronization points.

However, the recent introduction of the mesh shading pipeline ([Kubisch 2018; Moore 2023; Oberberger et al. 2023]) allowed for the use of more complex rendering primitives as long as they could be expressed as points, lines, or triangles at the end of the mesh shading stage. The mesh shading pipeline replaces the vertex, geometry, and tesselation stage with an optional task stage and mesh stage. The mesh stage resembles the compute stage and supports the same kind of inputs, but its outputs are geometric primitives that are directly fed to the rasterizer. Similar to the compute

Author's address: Thibault Tricard, thibault.tricad@inria.fr, Maverick, INPG, LJK, Inria, Grenoble, France.

shader, the mesh shader uses a workgroup to distribute complex operations on multiple threads that share parts of their memory. This grants more control over the geometry than the geometry and tesselation stages.

In the context of explicit surface rendering, mesh shading pipeline opened new possibilities for continuous level-of-detail approaches [Englert 2020], for on-the-fly tesselation [Santerre et al. 2020], and for early culling of geometry [Unterguggenberger et al. 2021] by using tailored meshlet generation strategies [Jensen et al. 2023]. Mesh shading can also be used to decompress meshlet in real time [Kuth et al. 2024]. Recently [Kreskowski et al. 2022] proposed to use mesh shaders to render implicit surfaces using rasterization rather than marching algorithms. While this greatly reduces the cost of isosurface rendering, this method cannot be used in volume rendering. However, to the best of our knowledge, no method has been proposed to process volume primitives.

In this article, we propose to use tetrahedrons as a rendering primitive. Tetrahedrons are already popular among the computer graphics community: they support being deformed [Gascon et al. 2013], and they are used in mechanical simulation [Koschier et al. 2014] and fluid simulation [Ando et al. 2013].

We propose a method that takes advantage of the new possibilities offered by the mesh shading pipeline to process tetrahedrons and use them to invoke ***Interval Shaders***. Here, we introduce an ***Interval Shader*** as a fragment shader that receives a depth interval for a single fragment. The first value corresponds to the depth of the front faces of the tetrahedron and the second to the back faces. The interval shader uses this depth interval to compute the current fragment's color for volume rendering. As a fragment can only have one depth on current hardware, we propose a method to process tetrahedrons using mesh shaders to hijack the rasterizer to compute two depths per fragment.

2 METHOD

In this section, we propose a method to invoke an interval shader using a mesh shader. First, we address the generation of depth intervals in a simple case with strict constraints (Section 2.1), then we show a method to process tetrahedrons so they match these constraints 2.2.

2.1 Generating Depth Intervals: Simple Case

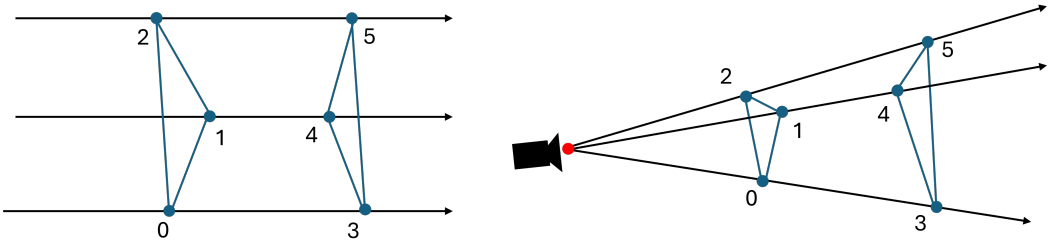

Fig. 2. Illustration of a prismoid in projected space (**left**), and in world space (**right**).

Let's consider a triangular prismoid [1] in projected space composed of six vertices v_n with $n \in [0, 5]$, with v_0, v_1, v_2 forming the first triangular base, and v_3, v_4, v_5 forming the second (see Figure 2). We place ourselves in the case where both bases of the prismoid project to a single triangle in screen space such that :

[1] a prism whose bases are not parallel

$$\frac{v_0.xy}{v_0.w} = \frac{v_3.xy}{v_3.w}, \ \frac{v_1.xy}{v_1.w} = \frac{v_4.xy}{v_4.w}, \ \frac{v_2.xy}{v_2.w} = \frac{v_5.xy}{v_5.w}$$

Any point p on this triangle is defined as follows :

$$\begin{aligned} p.xy &= \frac{v_0.xy}{v_0.w} * \lambda_0 + \frac{v_1.xy}{v_1.w} * \lambda_1 + \frac{v_2.xy}{v_2.w} * \lambda_2 \\ &= \frac{v_3.xy}{v_3.w} * \lambda_0 + \frac{v_4.xy}{v_4.w} * \lambda_1 + \frac{v_5.xy}{v_5.w} * \lambda_2 \end{aligned} \tag{1}$$

with λ_0, λ_1, and λ_2 being the **screen space barycentric coordinates** of the point p. Note that as we are using **screen space barycentric coordinates** we can use the same set for the first and second bases of the prism. We then find the z coordinates of p using:

$$\frac{1}{p.z_0} = \frac{v_0.w}{v_0.z} * \lambda_0 + \frac{v_1.w}{v_1.z} * \lambda_1 + \frac{v_2.w}{v_2.z} * \lambda_2 \tag{2}$$

$$\frac{1}{p.z_1} = \frac{v3.w}{v_3.z} * \lambda_0 + \frac{v_4.w}{v_4.z} * \lambda_1 + \frac{v_5.w}{v_5.z} * \lambda_2 \tag{3}$$

with $p.z_0$ the z coordinate of the point p on the first base of the prism and $p.z_1$ the z coordinate of the point p on the second. We finally obtain, $z_{min} = min(p.z_0, p.z_1)$ and $z_{max} = max(p.z_0, p.z_1)$.

Now that we know how to compute z_{min} and z_{max}, we propose to hijack the rasterizer into computing them for us, so as to have them as input variables in the fragment stage. To do so, we emit a triangle proxy for the prismoid with the following vertices coordinates:

$$\left\{\frac{v_0.x}{v_0.w}, \frac{v_0.y}{v_0.w}, 0, 1\right\}, \left\{\frac{v_1.x}{v_1.w}, \frac{v_1.y}{v_1.w}, 0, 1\right\}, \left\{\frac{v_2.x}{v_2.w}, \frac{v_2.y}{v_2.w}, 0, 1\right\}$$

and with the following vertices attributes Z:

$$\left\{\frac{v_0.w}{v_0.z}, \frac{v_3.w}{v_3.z}\right\}, \left\{\frac{v_1.w}{v_1.z}, \frac{v_4.w}{v_4.z}\right\}, \left\{\frac{v_2.w}{v_2.z}, \frac{v_5.w}{v_5.z}\right\}$$

By setting the z coordinates to 0 we force the rasterizer to use the **screen space barycentric coordinates** to interpolate the vertices attributes. The interpolation of the Z by the **screen space barycentric coordinates** is equivalent to the equation 2 and 3. Thus we have $1/z_{min}$ and $1/z_{max}$ as input of the fragment shader. We can now use this interval for our shading.

2.2 Tetrahedrons

This approach is only valid for a prismoid whose bases project onto a single triangle in screen space, or for a shape that has been decomposed into such prismoids. While a prismoid decomposition would be a complex task in a general case, in the specific case of a tetrahedron, this operation can be solved analytically in real time in a mesh shader. For a tetrahedron, there are only two cases to consider (Figure 3):

- Case 1: when the tetrahedron projects onto a single triangle in screen space, the decomposition creates three triangle prismoids.
- Case 2: when the tetrahedron projects onto a quad in screen space, the decomposition creates four triangle prismoids.

Specific cases where one or two faces of the tetrahedron have a null area in screen space can be handled by the first case with a numerically stable implementation.

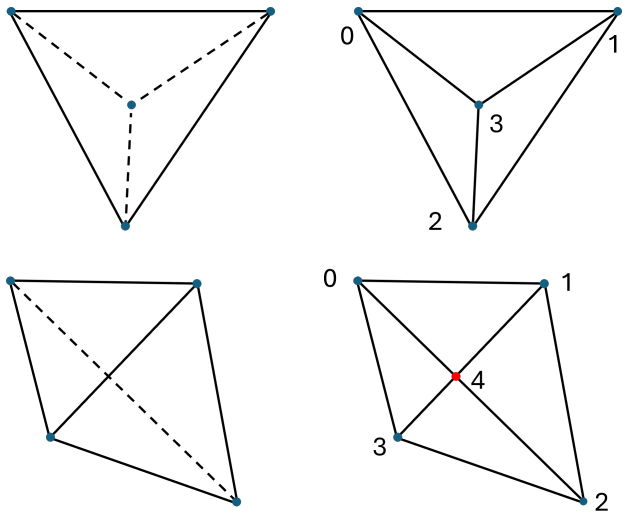

Fig. 3. Illustration of two possible cases of the prismoid decomposition for a tetrahedron. **left**: Wireframe of the tetrahedron. **right**: Proxy generated by the mesh shader with the vertex indices used. **top**: First case, three triangles and and four vertices are created. **bottom**: Second case, four triangles and five vertices are created.

2.2.1 Emitting the proxy. As, in Section 2.1 we emitted one triangle proxy for the prismoid we processed, here we need to emit proxies composed of as many triangles as we have prismoid: 3 in case 1 and 4 in case 2. To create the proxies we must compute the position of all required points in screen space, and their depth**s** in the projected space. We start by defining t_i the i^{th} vertices of the tetrahedron with $i \in \{0, 3\}$, and p_j the j^{th} vertices of the proxy.

Case 1. In this case, we define $p_{0:2}$ as the points on the silhouette of the proxy and p_3 as the point not on the silhouette. With:

$$p_{0:3} = t_{a:d}.xy/t_{a:d}.w$$

With *a*,*b*,*c*, and *d* the indices of the vertices on the tetrahedron, chosen such that $p_{0:2}$ are ordered in a clockwise manner as shown in Figure 3. For all vertices $p_{0:2}$ on the silhouette, we define:

$$z_{min0:2} = z_{max0:2} = \frac{t_{a:c}.z}{t_{a:c}.w}$$

For the vertices p_3, we compute its screen space barycentric coordinates λ on the triangle $p_{0:2}$ using the edge function [Pineda 1988] such that:

$$p_3.xy = \frac{t_a.xy}{t_a.w} * \lambda_0 + \frac{t_b.xy}{t_b.w} * \lambda_1 + \frac{t_c.xy}{t_c.w} * \lambda_2$$

Then we project back p_3 on both the front and back faces to find its depths. As p_3 is the screen space projection of t_d we find:

$$z_0 = t_d.z/t_d.w$$

To find the second projection of p_3 on either the front face or back face: we follow the Equation 2:

$$\frac{1}{z_1} = \frac{t_a.w}{t_a.z} * \lambda_0 + \frac{t_b.w}{t_b.z} * \lambda_1 + \frac{t_c.w}{t_c.z} * \lambda_2$$

Then we have $z_{min} = min(z_0, z_1)$ and $z_{max} = max(z_0, z_1)$.

Finally, we emit three triangles proxies as described in Section 2.1: $\{p_0, p_1, p_3\}$, $\{p_1, p_2, p_3\}$, and $\{p_2, p_0, p_3\}$ (see Figure 3).

Case 2. In this case, we start by defining $p_{0:3}$ as the point on the silhouette of the proxy and p_4 as the point not on the silhouette. Similarly, as in the first case, we define:

$$p_{0:3} = t_{a:d}.xy / t_{a:d}.w$$

and:

$$z_{min\,0:3} = z_{max\,0:3} = \frac{t_{a:d}.z}{t_{a:d}.w}$$

with a,b,c, and d the indices of a vertex on the tetrahedron chosen such that $p_{0:3}$ are ordered in a clockwise manner as shown in Figure 3. We can find p_4 by solving the intersection of the segment p_0p_2, and p_1p_3 such that:

$$p_4 = p_0 + (p_2 - p_0) * t = p_1 + (p_3 - p_1) * s$$

with t and, respectively, s, the linear interpolation weight on the segment p_0p_2, and respectively p_1p_3. We adapt Equation 3 to segments to find z_0 and z_1 giving us:

$$\frac{1}{z_1} = \frac{t_a.w}{t_a.z} * (1 - t) + \frac{t_c.w}{t_c.z} * t$$

and:

$$\frac{1}{z_0} = \frac{t_b.w}{t_b.z} * (1 - s) + \frac{t_d.w}{t_d.z} * s$$

Then we have $z_{min4} = min(z_0, z_1)$ and $z_{max4} = max(z_0, z_1)$. Finally, we emit four proxies as described in Section 2.1: $\{p_0, p_1, p_4\}$, $\{p_1, p_2, p_4\}$, $\{p_2, p_3, p_4\}$, and $\{p_3, p_0, p_4\}$ (see Figure 3).

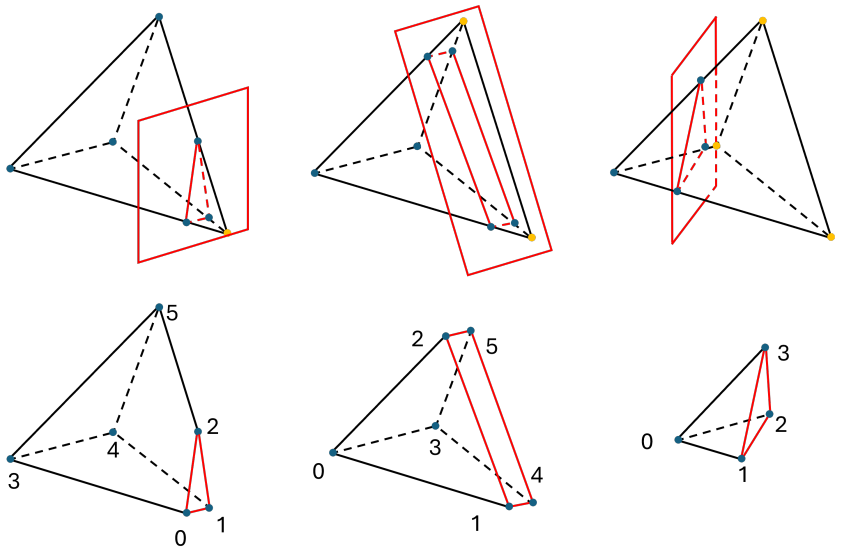

Fig. 4. Clipping of a tetrahedron by a plane. **Top**: the tetrahedron, the clipping plane, and its intersection with the tetrahedron in red, in yellow the clipped vertices. **Bottom**: The clipped tetrahedron and the indices we use to describe them. **Left**: one vertex clipped by the plane. **Center**: two vertices clipped by the plane. **Right**: three vertices clipped by the plane.

2.3 Clipping

As explained in Section 2.1, when creating the proxy, we store the inverse of the depths as vertex attributes. Thus, if the z_{min} value gets too close to zero (when the tetrahedron is too close to the near plane), the interpolation $1/z_{min}$ will create numerical instability. To avoid this effect, we propose to clip the tetrahedrons before the near plane (at $n_e = near + \epsilon$ in camera space). Depending on the number of vertices having a z coordinate inferior to n_e this operation can create two shapes: either a prism-like shape (two triangles, three parallelograms), or a tetrahedron (see Figure 4). If the output is a prism-like shape we decompose it into three tetrahedrons and process them as described in Section 2.2. There are multiple valid ways to split this shape into tetrahedrons. Given the indexing shown in Figure 4 we split it as follows: $\{0, 1, 2, 3\}$, $\{1, 2, 3, 5\}$, $\{1, 3, 4, 5\}$. In addition to avoiding unexpected behavior, clipping the tetrahedron before the near plane allows us to place our camera inside a (or a set of) tetrahedron, as can be seen in Figure 8.

3 RESULTS

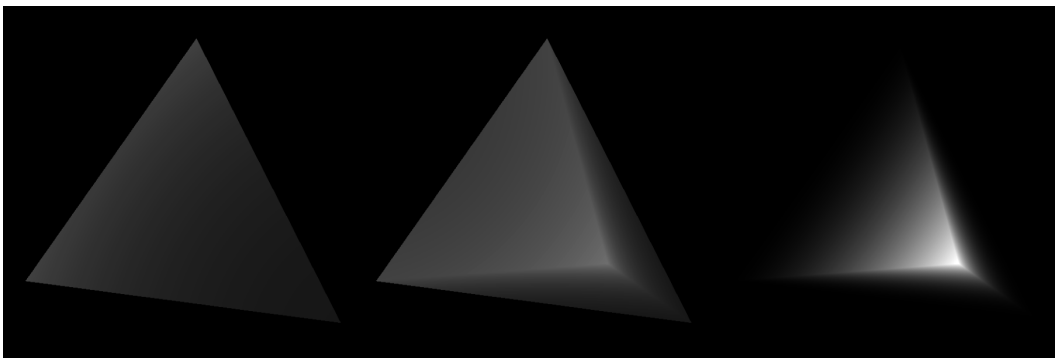

Fig. 5. Single tetrahedron processed by our method. **Left**: depth of the front face, **Center**: depth of the back faces, **Right**: difference in depth in world space

In this Section, we show some results obtained with our method. Figure 5 shows the result of our method while dealing with a single tetrahedron. Figure 5 Left shows the depth of the front faces, Figure 5 Center shows the depth of the back faces, and Figure 5 Right shows the difference in depth between the front and the back faces. This computation is done in one draw call as the fragment shader receives the depth of the front and back faces. This allows us to exploit the depth intervals in two main ways :

- To compute the optical depth of a tetrahedral mesh as described in Section 3.1
- As bounds for a marching algorithm as described in Section 3.2

3.1 Optical Depth

Figure 6 demonstrates how our method can be used to compute the optical depth in tetrahedral meshes. To achieve this result we chose a mesh from cgtrader [The-Ni11 2018] and converted it into a tetrahedral mesh using the Geogram library [Levy 2015]. The resulting mesh was directly fed as an indexed tetrahedron list to our mesh shaders without modification or reordering. Then in the fragment shader, given the difference in depth generated for each fragment, we compute the position of the front and back faces in world space, and output the norm of their differences. Following that, we rely on the blending phase of the pipeline to sum those lengths (using the add blend function and blending parameter equal to one) which gives us the distance spent in the

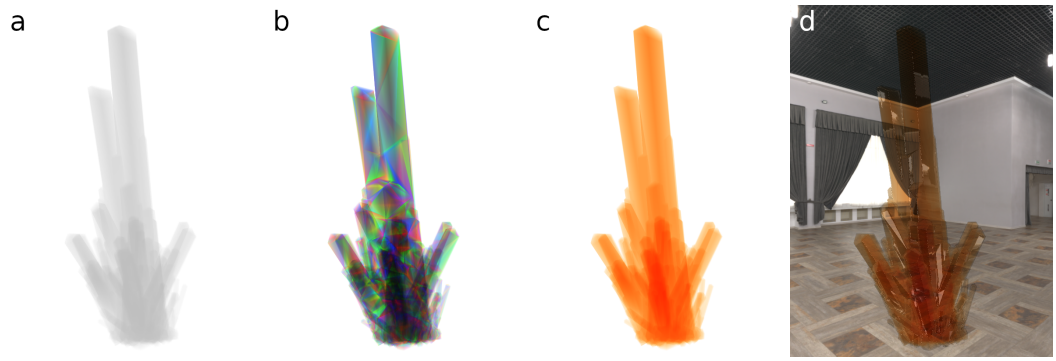

Fig. 6. Crystal rendered using our method. **a**: sum of interval length. **b**: Light transmittance where each tetrahedron has a different absorption color. **b**: Light transmittance where each tetrahedron absorbs the same color. **d**: Light transmittance and reflection based on an environment map.

model for each pixel. Finally, we apply the Beer-Lambert law as a post-process to evaluate the light transmitted by the model.

This allows us to evaluate the transmittance in any tetrahedron mesh in one draw call followed by a simple post-process. Our method is not limited by the number of intersections between the camera ray and the surface as opposed to A-Buffer-like approaches [Everitt 2001] and gives exact results.

3.2 Ray Marching

Our method can also be used to render signed distance fields. In Figure 7 we show an asteroid field ray-marched with a sphere tracing algorithm [Hart 1996]. To achieve this we instantiate multiple tetrahedrons, each bounding a procedural signed distance field defined using Hypertextures [Perlin and Hoffert 1989] in a frame centered on the tetrahedron.

In this case, each fragment only computes the ray-SDF intersection for only one asteroid, and we rely on the depth buffer to choose which fragment to show on screen. This allows us to distribute the computation of the ray-SDF intersection on multiple fragments per ray, thus limiting the complexity of the marching done in each fragment, but preventing us from stopping the marching at the first intersection. As, in most cases, the cost of an image is determined by the most expensive fragment, splitting the computation of one ray into multiple fragments reduces the overall cost of the rendering. This is particularly useful in the case of sphere tracing [Hart 1996] where the cost of a fragment increases greatly for grazing rays. Here the worst-case scenario would be rays that closely miss multiple surfaces. Our method distributes those close misses on various fragments that can be executed in parallel depending on warp availability.

4 PERFORMANCES

All our examples have been rendered on an NVIDIA GeForce RTX 4080 and are rendered in 4K resolution (3 840 × 2 160) using Vulkan 1.3.242. Each mesh shading work group processes one input tetrahedron, using one thread per work group. The performances of our method are described in Table 1. Our measure shows that our method remains real-time even when used to invoke ray marching (see Section 3.2). When comparing the rendering time of the crystal to the asteroids that use a similar amount of tetrahedrons we can see that most of the time is spent in the ray marching algorithm. Our example, in Figure 8, shows that our method can handle large amounts of primitive while remaining real-time even when clipping is required.

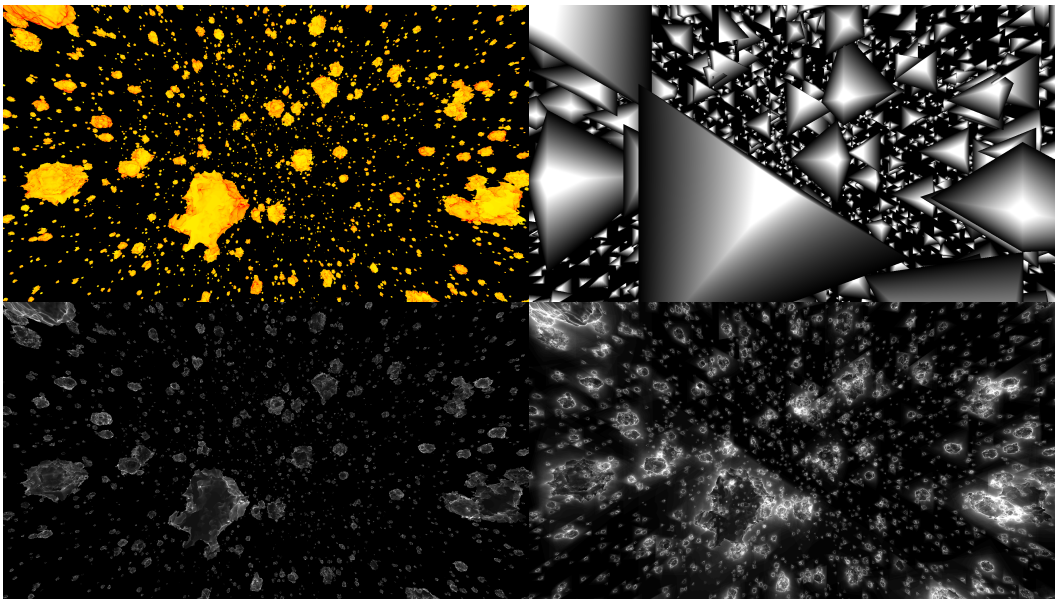

Fig. 7. Field of asteroids evaluated with our method inspired by [Neyret 2024]. **Top left**: asteroid shaded using the number of steps of sphere marching before reaching the surface. **Top right**: rendering of the bounding tetrahedron shaded using the interval length to march. **Bottom left**: cost of the visible fragment (black means 0 sphere marching steps, white means 60 sphere marching steps). **Bottom right**: sum of the cost of all fragments evaluated (black means 0 sphere marching steps, white means 60 sphere marching steps).

Model	Figure	tetrahedron count	cost (ms)
Bunny	1	61 042	3.30
Single tetrahedron	5	1	0.17
Crystal	6	8 580	0.71
Asteroids	7	10 000	7.45
Armadillo	8 a	959 042	3.88
Armadillo	8 b	959 042	11.8
Armadillo	8 c	959 042	12.7

Table 1. Performances measures of our examples

5 LIMITATIONS

The method we propose allows us to generate shading intervals that can be evaluated in parallel and then recombined in the blending phase. As a result, it can only be applied to render volume information that is not order-dependent.

Our method uses screen space proxies to force the rasterizer to use screen space barycentric coordinates to interpolate depths correctly. In consequence, the interpolation of vertice attributes cannot be computed correctly by the rasterizer. Hence, our methods can not be applied to render tetrahedrons with vertex attributes, only per tetrahedron information can be used for the shading. An example of this can be shown in Figure 6 Center Left where each tetrahedron has a different absorption color, and in Figure 7 where each tetrahedron has a different model matrix that is inverted in the fragment shader.

Currently, our method can only handle one tetrahedron per mesh shading work group which limits the number of tetrahedrons that can be drawn at the same time. Attempts at processing more tetrahedrons per work group resulted in lower performances either due to branch divergence in the mesh shader or to inconsistent memory access when fetching tetrahedrons. This limitation could potentially be addressed by reorganizing tetrahedrons in meshlet-like structures.

6 CONCLUSION

We proposed to use tetrahedron as a volume rendering primitive and a pipeline to rasterize them. This allows for rendering transparent objects accurately in a single pass and for invoking the ray marching algorithm with marching intervals. Our method can render volumic data as long as we can bind them by one or multiple tetrahedrons and their rendering can be done in an unordered fashion. We believe our method can open new possibilities for volume rendering. Moreover, if our method for processing tetrahedrons is implemented in future hardware it could widen the possibility for interval shading and volume rendering in general.

REFERENCES

Erik W. Anderson, Steven P. Callahan, Carlos E. Scheidegger, John M. Schreiner, and Claudio T. Silva. 2007. Hardware-Assisted Point-Based Volume Rendering of Tetrahedral Meshes. In *XX Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI 2007)*. 163–172. https://doi.org/10.1109/SIBGRAPI.2007.20

Ryoichi Ando, Nils Thürey, and Chris Wojtan. 2013. Highly adaptive liquid simulations on tetrahedral meshes. *ACM Trans. Graph.* 32, 4, Article 103 (jul 2013), 10 pages. https://doi.org/10.1145/2461912.2461982

Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. 2008. Interactive multiple anisotropic scattering in clouds. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games* (Redwood City, California) *(I3D '08)*. Association for Computing Machinery, New York, NY, USA, 173–182. https://doi.org/10.1145/1342250.1342277

Deukhyun Cha, Sungjin Son, and Insung Ihm. 2009. GPU-Assisted High Quality Particle Rendering. *Computer Graphics Forum* 28, 4 (2009), 1247–1255. https://doi.org/10.1111/j.1467-8659.2009.01502.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01502.x

Matthias Englert. 2020. Using Mesh Shaders for Continuous Level-of-Detail Terrain Rendering. In *ACM SIGGRAPH 2020 Talks* (Virtual Event, USA) *(SIGGRAPH '20)*. Association for Computing Machinery, New York, NY, USA, Article 44, 2 pages. https://doi.org/10.1145/3388767.3407391

Cass W. Everitt. 2001. Interactive Order-Independent Transparency. https://api.semanticscholar.org/CorpusID:5813703

Jorge Gascon, Jose M. Espadero, Alvaro G. Perez, Rosell Torres, and Miguel A. Otaduy. 2013. Fast deformation of volume data using tetrahedral mesh rasterization. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Anaheim, California) *(SCA '13)*. Association for Computing Machinery, New York, NY, USA, 181–185. https://doi.org/10.1145/2485895.2485917

John C. Hart. 1996. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer* 12, 10 (01 Dec 1996), 527–545. https://doi.org/10.1007/s003710050084

J. Huang and M.B. Carter. 2005. Interactive transparency rendering for large CAD models. *IEEE Transactions on Visualization and Computer Graphics* 11, 5 (2005), 584–595. https://doi.org/10.1109/TVCG.2005.82

Mark Bo Jensen, Jeppe Revall Frisvad, and J. Andreas Bærentzen. 2023. Performance Comparison of Meshlet Generation Strategies. *Journal of Computer Graphics Techniques (JCGT)* 12, 2 (8 12 2023), 1–27. http://jcgt.org/published/0012/02/01/

Dan Koschier, Sebastian Lipponer, and Jan Bender. 2014. Adaptive Tetrahedral Meshes for Brittle Fracture Simulation. In *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*, Vladlen Koltun and Eftychios Sifakis (Eds.). The Eurographics Association. https://doi.org/10.2312/sca.20141123

A. Kreskowski, G. Rendle, and B. Froehlich. 2022. Efficient Direct Isosurface Rasterization of Scalar Volumes. *Computer Graphics Forum* 41, 7 (2022), 215–226. https://doi.org/10.1111/cgf.14670 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14670

Christoph Kubisch. 2018. Introduction to Turing Mesh Shaders. https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/.

Bastian Kuth, Max Oberberger, Felix Kawala, Sander Reitter, Sebastian Michel, Matthäus Chajdas, and Quirin Meyer. 2024. Towards Practical Meshlet Compression. arXiv:2404.06359 [cs.GR]

Bruno Levy. 2015. Geogram. *GitHub repository. URL: https://github.com/BrunoLevy/geogram* (2015).

Warren Moore. 2023. Mesh Shaders and Meshlet Culling in Metal 3. https://metalbyexample.com/mesh-shaders/.

Fabrice Neyret. 2024. peaky/floffy Perlin hypertexture. https://www.shadertoy.com/view/XcK3RW.

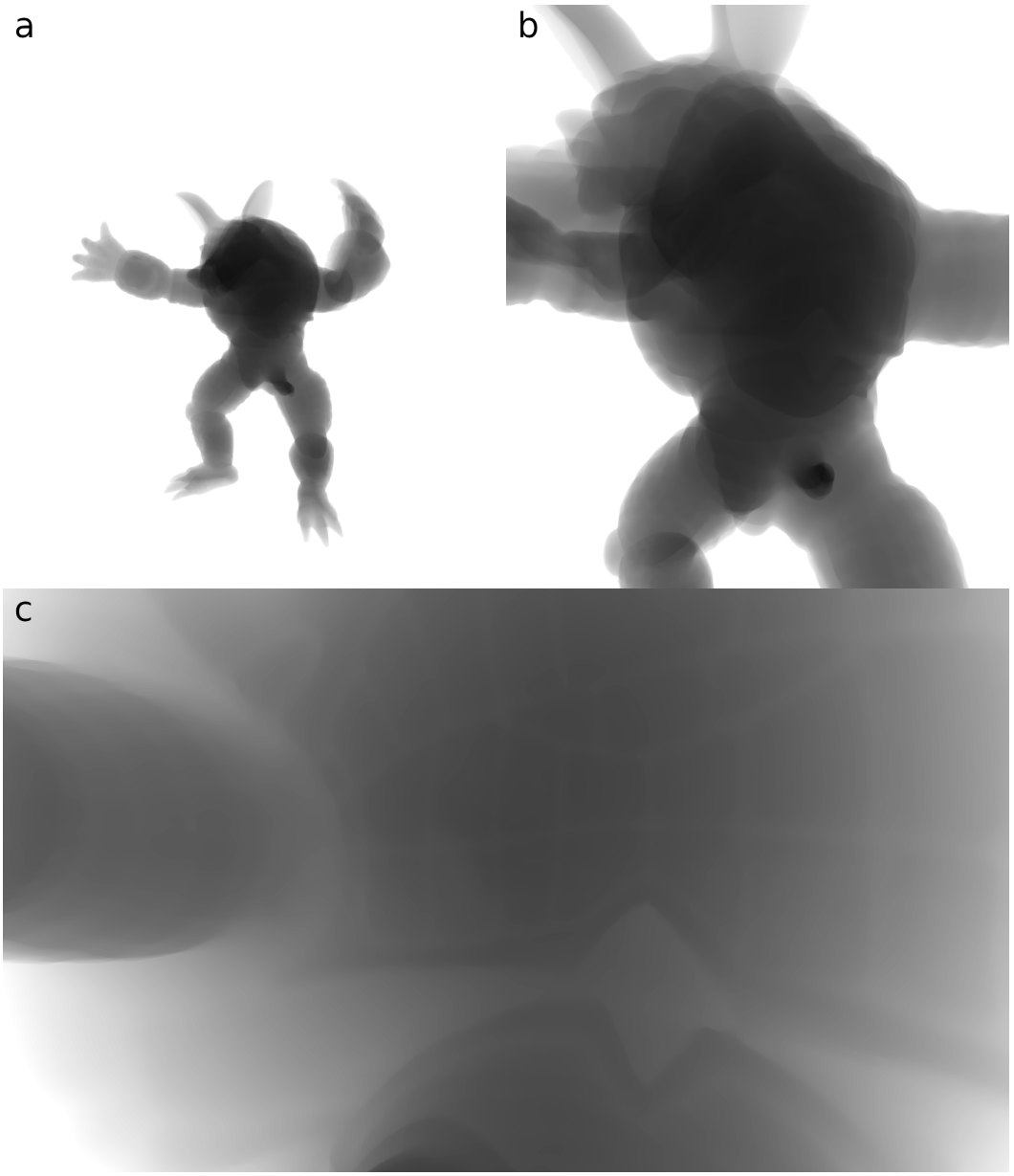

Fig. 8. Rendering of the tetrahedrized Armadillo with the camera progressively moving forward. **a**: the camera is far away. **b**: the camera is close but outside of the model. **c**: the camera is inside the model.

Max Oberberger, Bastian Kuth, and Quirin Meyer. 2023. From vertex shader to mesh shader. https://gpuopen.com/learn/mesh_shaders/mesh_shaders-from_vertex_shader_to_mesh_shader/.

K. Perlin and E. M. Hoffert. 1989. Hypertexture. *SIGGRAPH Comput. Graph.* 23, 3 (jul 1989), 253–262. https://doi.org/10.1145/74334.74359

Juan Pineda. 1988. A parallel algorithm for polygon rasterization. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '88).* Association for Computing Machinery, New York, NY, USA, 17–20.

https://doi.org/10.1145/54852.378457

Benjamin Santerre, Masaki Abe, and Taichi Watanabe. 2020. Improving GPU Real-Time Wide Terrain Tessellation Using the New Mesh Shader Pipeline. In *2020 Nicograph International (NicoInt)*. 86–89. https://doi.org/10.1109/NicoInt50878.2020.00025

The-Ni11. 2018. https://www.cgtrader.com/free-3d-models/architectural/decoration/red-crystal-dc79c122-8a2e-48ee-aa07-1d4c96528151.

Johannes Unterguggenberger, Bernhard Kerbl, J. Pernsteiner, and M. Wimmer. 2021. Conservative Meshlet Bounds for Robust Culling of Skinned Meshes. *Computer Graphics Forum* 40 (10 2021), 57–69. https://doi.org/10.1111/cgf.14401