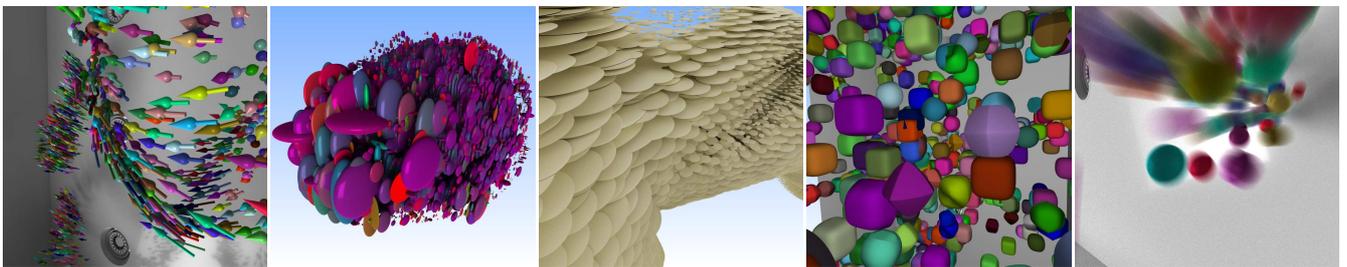


# High-Quality Rendering of Glyphs Using Hardware-Accelerated Ray Tracing

S. Zellmann<sup>1</sup> , M. Aumüller<sup>2</sup> , N. Marshak<sup>3</sup>  and I. Wald<sup>4</sup> 

<sup>1</sup>University of Cologne, Chair of Computer Science

<sup>2</sup>University of Stuttgart, HLRS <sup>3</sup>University of Utah, SCI <sup>4</sup>NVIDIA



**Figure 1:** Glyph visualizations with several different shapes and appearance. Left: Particle flow in a combustion chamber visualized with arrow glyphs and high-quality rendering with path tracing and filmic BRDF. Second from left: Diffusion tensor imaging with tensor eigenvalues mapped as RGB colors. Middle: Diffusion tensor imaging rendered with ambient occlusion to help with clutter. Second from right: Isotropic superquadric glyphs. Right: Motion blur to provide additional visual cues.

## Abstract

Glyph rendering is an important scientific visualization technique for 3D, time-varying simulation data and for higher-dimensional data in general. Though conceptually simple, there are several different challenges when realizing glyph rendering on top of triangle rasterization APIs, such as possibly prohibitive polygon counts, limitations of what shapes can be used for the glyphs, issues with visual clutter, etc. In this paper, we investigate the use of hardware ray tracing for high-quality, high-performance glyph rendering, and show that this not only leads to a more flexible and often more elegant solution for dealing with number and shape of glyphs, but that this can also help address visual clutter, and even provide additional visual cues that can enhance understanding of the dataset.

## CCS Concepts

• *Human-centered computing* → *Scientific visualization; Visualization techniques*; • *Computing methodologies* → *Ray tracing; Graphics processors*;

## 1. Introduction

Glyph-based rendering is a popular scientific visualization technique and is traditionally implemented with rasterization, point splatting, or ray casting of implicit surfaces. Each of these techniques have their individual merits but also challenges. Purely rasterization-based approaches e.g. require the glyphs to be tessellated, which limits the complexity and the number of the shapes being used. While ray tracing glyphs represented with implicit surfaces allows for high-quality images and visually pleasing results, calculating intersections involves costly root finding algorithms.

The RTX ray tracing cores found on current-generation NVIDIA

GPUs can perform ray / primitive intersection tests with hardware-accelerated bounding volume hierarchy (BVH) traversal and support hardware instancing as well as user-defined primitives. APIs that expose the RTX hardware extension are OptiX [PBD\*10], Vulkan [NVI18] or Microsoft DXR [Mic18].

With ray tracing hardware being available even on consumer graphic cards, we argue that ray tracing is a viable, if not superior, option to implement scientific visualization algorithms and demonstrate this using glyph rendering. Ray tracing opens the door for techniques that could not easily be implemented with traditional rasterization-based approaches. In this paper we contribute:

- An implementation of various glyph rendering techniques using hardware-accelerated ray tracing with RTX.
- Examples of how ray tracing-based algorithms that make use of global illumination effects can deal with visual clutter that is typical for glyph data sets.
- Examples of rendering techniques that are typically implemented with ray tracing, which cannot easily be reproduced with rasterization but can help to provide additional visual cues.

## 2. Related Work

Glyphs are typically used for medical [MRZH14] or particle flow visualization [GRE09, RGE19] and are often implemented using rasterization hardware [TL04]. They can be classified by shape (e.g. deformed spheres, superquadrics, or arrows) and appearance (e.g. color or transparency) [ROP11].

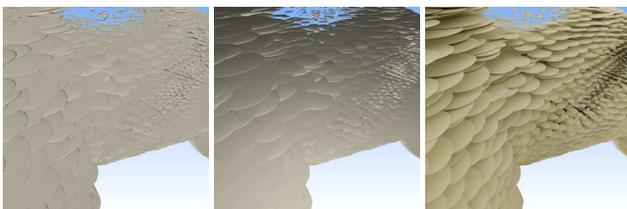
In the context of diffusion tensor imaging (DTI), tensor parameters are mapped as anisotropy to the parameters of superquadric glyphs [Kin04]. Ropinski et al. [RSMS\*07] use superquadrics to represent the principal eigenvectors of diffusion tensors. Schultz and Kindlmann [SK10] extend the scope of glyphs to indefinite tensors with negative eigenvalues by using concave superquadrics, and Gerrits et al. [GRT16] lift the restriction to symmetric tensors. Podlozhnyuk et al. [PPK17] present a C++ implementation of superquadric evaluation using Newton’s root finding method.

Several rendering systems are aimed at high-quality scientific visualization based on ray tracing [WJA\*17, ZHL17] or extend medical visualization algorithms with ray tracing [ZWL17]. The recent success of ray tracing for interactive visualization applications can be attributed to the existence of ray tracing kernel libraries [PBD\*10, WWB\*14] that provide optimized implementations for ray / primitive intersection.

The introduction of RTX hardware has led to several research papers. Ganter and Manzke [GM19] as well as Morrical et al [MUWPI19] use RTX for volume rendering with empty space skipping. Wald et al. [WUM\*19] use RTX for tetrahedron point queries and thus for an application that is not limited to rendering.

## 3. Method Overview

We present a prototypical glyph rendering system with RTX that supports a variety of glyph types with different shape and appearance plus static triangle base geometry (see Figure 1). Our system uses OptiX 7 and the high-level wrapper library owl [WMH20].



**Figure 2:** Mechanical engineering data set with 280 K instanced spheres. Left: OpenGL rendering. Middle: primary ray casting. Right: path tracing with an omnidirectional light, where ambient occlusion helps to significantly reduce visual clutter.

### 3.1. Geometry setup

RTX accelerates ray / object intersections using BVH traversal and triangle intersection in hardware. Custom primitives can be added via *intersect* programs that run on the GPU shader cores and thus require context switches. Traversal can be intercepted using *closest-hit* and *any-hit* programs. The user can alter the ray intersection parameter or reject the intersection from within those programs. RTX allows us to define a two-level hierarchy where the bottom level is expressed using instance transforms. When traversing an instance, the world space ray is transformed by the inverse instance transform. This allows for cheaply creating lots of copies of one object.

We distinguish between *affine glyphs* that can be represented as an affine copy of a template geometry (e.g. spheres that transform to ellipsoids) and other *non-affine glyphs* (e.g. arrow glyphs with fixed-length arrow heads, which cannot be scaled non-uniformly without objectionable distortion). Affine glyphs can be efficiently implemented using a custom intersect program and a two-level BVH, where a *single base geometry* is spread out all over the space using transforms. Non-affine glyphs must be implemented by using one base geometry per glyph instance. This can still be beneficial as the instance bounds tightly bound the complex glyph shape.

### 3.2. Rendering setup

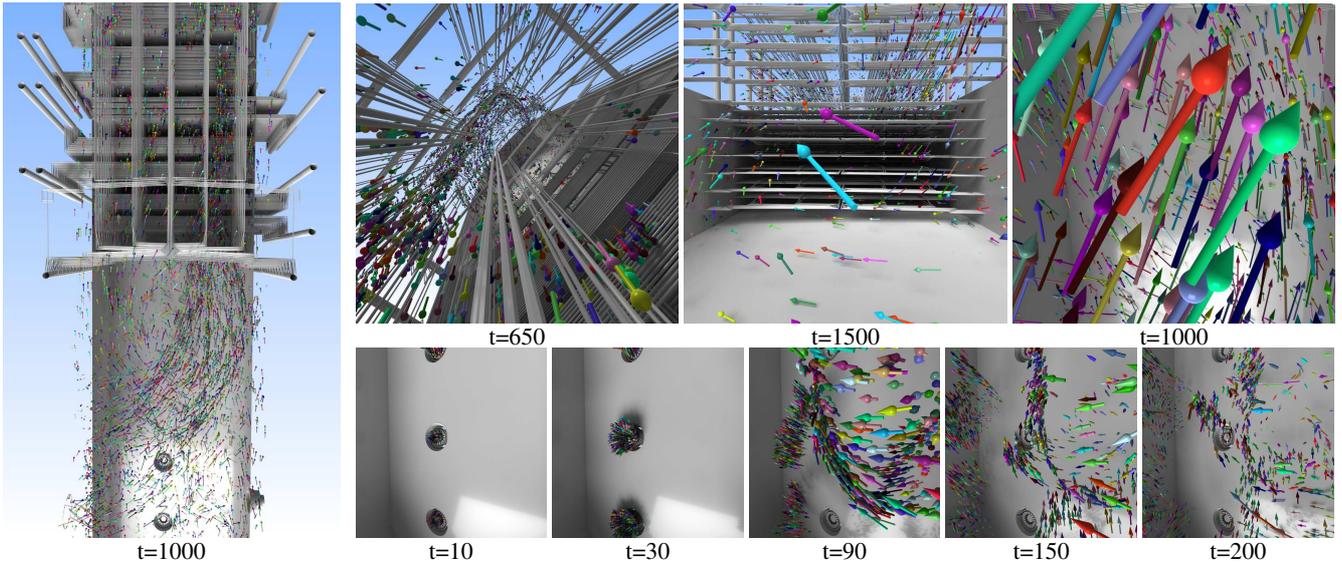
We support interactive rendering using local shading with primary rays and high-quality shading with naïve path tracing and an omnidirectional light source. With the latter we render convergence frames that on their own are very noisy but through accumulation gradually converge to a high-quality image. At any point during the interactive visualization, we can apply tone mapping to the accumulation buffer and write its content to the framebuffer for interactive display. High visual fidelity is achieved by using Usher’s implementation of the Disney BRDF [MHH\*12, Ush19].

### 3.3. Particle flow visualization with glyphs

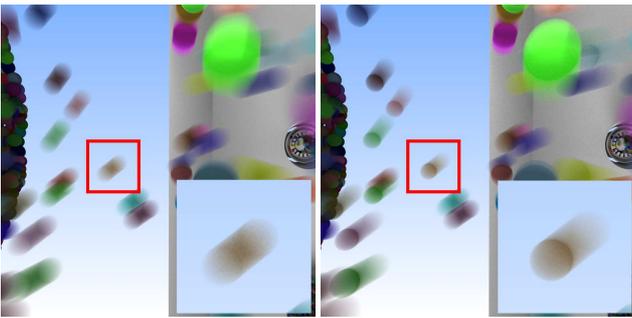
With particle flow data, we keep all timesteps in CPU main memory and synchronously upload them to the GPU on demand. We choose arrow glyphs as those can be fully represented with quadric surfaces (capped cylinder and rounded cone for the arrow head, see Figure 3), so that zooming in will not reveal tessellation. In order to retain the world-space proportions of the arrow heads, we store the unit size glyph geometry in a GPU buffer and allow RTX to transform it by building a two-level BVH with instance transforms. Uploading animation frames—comprised of one affine transform per glyph, and the glyph geometry itself—comes at moderate memory transfer costs. Reuploading the data also requires us to rebuild the BVH on the GPU, which takes on the order of ten milliseconds.

### 3.4. Lighting models to reduce visual clutter

Glyph data is known to be prone to visual clutter [RSMS\*07]. Rendering with local illumination only—the default mode in scientific visualization systems like ParaView [AGL05] or VisIt [CBW\*12]—makes it hard to discern visual features. Path-traced global illumination with its implicitly generated ambient occlusion shadows can help to reduce that visual clutter and provides



**Figure 3:** Fully converged high-quality images of a time-varying combustion simulation in a coal power plant. For this model we combine arrow glyph rendering with a static triangle geometry (657 K triangles). The data set counts in at 2000 timesteps and 12500 particles in most timesteps. With RTX, we can render convergence frames at interactive rates for this viewport of  $800 \times 800$  pixels.

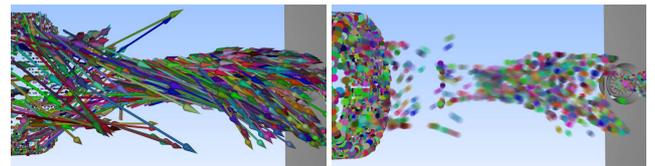


**Figure 4:** Left: Motion blur rendering of spheres based on velocity only. Since the blur is equally spread out, we gain insight on speed and direction of particle advection, but not on the absolute orientation. Right: Factoring acceleration into the calculation provides us with this additional cue—blurriness increases with velocity.

additional depth cues. Figure 2 shows an example where local shading is compared to ambient occlusion from path tracing. As ambient occlusion uses an omni-directional light source, frames will generally converge at interactive rates.

### 3.5. Adding visual cues with motion blur

A technique that is very common in the movie industry for filmic rendering and that can be very elegantly implemented with ray tracing is motion blur [CPC84]. This technique models the eponymous aperture error when sampling a moving scene with respect to time: instead of capturing an exact representation at a precise moment, an average over a short time interval is shown. Moving objects will appear spread out and blurred over the space they cross while the camera shutter is open. In filmic rendering, motion blur will pri-



**Figure 5:** Injection of fuel particles depicted as arrow glyphs (left) or spheres with motion blur (right). The sphere representation is less cluttered, more easily accessible to non-experts and succeeds better in showing that particles are injected at high velocity, but slow down quickly after entering the combustion chamber.

marily be used to simulate this effect, which essentially is just a deficiency of the underlying camera system.

As time-varying data sets contain enough information to actually simulate motion blur (cf. Figure 4), we propose motion blur as an efficient tool for depicting differences in speed. It is a representation that is easily understood by a non-technical audience. Furthermore, as motion is already conveyed by the blur, a simpler glyph can be used to encode a particle. Spherical particles with motion blur, for example, introduce less cluttering than a representation of movement by arrows of varying length (cf. Figure 5). A distributed ray tracer lends itself to implementing this effect: It is sufficient to make the particle position adhere to a stochastic distribution dependent on time.

### 3.6. Superquadrics

We support rendering superquadric surfaces in their parametric form  $|\frac{x}{A}|^r + |\frac{y}{B}|^s + |\frac{z}{C}|^t = 1$ . We use Newton's method for root finding. Instead of the obvious bounding box or sphere intersection, we decided to compute a coarse tessellation fully including the glyph and use the hardware-accelerated intersection with that as an initial



GPU	RTX		RTX		RTX		RTX	
	2080	4000	2080	4000	2080	4000	2080	4000
Primary Path Trace	197.	186.	111.	142.	76.8	111.	76.3	101.
OpenGL 6 × 12	147.	217.	160.	209.	226.	207.	218.	203.
12 × 12	72.8	106.	81.4	103.	145.	101.	132.	101.
24 × 12	37.1	55.7	40.4	54.7	75.1	53.6	66.2	53.4
24 × 24	19.0	29.0	20.7	28.5	38.2	28.1	33.4	28.0

**Table 1:** Frames / second for 280 K spheres and a 1200 × 1200 viewport. The OpenGL tessellation level is reported by the number of quadrilateral patches. Path tracing employs up to 10 bounces.

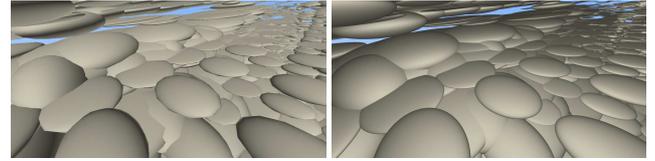
guess for the root. We refine the root using an any-hit program and report an intersection when the Newton refinement was successful.

### 3.7. Implementation

The implementation consists of C++ host programs and OptiX device programs for the various glyph modes. The host programs perform geometry setup and upload the instances as OptiX geometries and instance groups:

```
void AffineGeomHostProgram() {
    // Build single geometry and BVH accelerator
    OWLGeomType affineUserType = setupAffineType();
    OWLGeom geom = owlGeomCreate(affineUserType);
    OWLGroup grp = owlUserGeomGroupCreate(geom);
    owlGroupBuildAccel(grp);
    ...
    // Create top level
    for (Particle p : particles) {
        // Reuse the group from above
        owlInstanceGroupSetChild(world, p.idx, grp);
        // Set instance transform
        owlInstanceGroupSetTransform(world, p.idx, p.trans);
    }
    // Build top level BVH
    owlGroupBuildAccel(world);
}
```

```
void NonAffineGeomHostProgram() {
    // Build a template geometry for replication
    OWLGeomType nonAffineUserType = setupNonAffineType();
    OWLGeom geom = owlGeomCreate(nonAffineUserType);
    // The "real", non-affine geometry, accessed on the
    // device using the geometry instance's instID
    owlGeomSetBuffer(geom, geometryBuffer);
    ...
    // Create top level
    for (Particle p : particles) {
        // Build an individual accelerator per geometry
        OWLGroup grp = owlUserGeomGroupCreate(geom);
        owlGroupBuildAccel(grp);
        owlInstanceGroupSetChild(world, p.idx, grp);
        // Set instance transform
        owlInstanceGroupSetTransform(world, p.idx, p.trans);
    }
    // Build top level BVH
    owlGroupBuildAccel(world);
}
```



**Figure 6:** OpenGL with tessellation level 12 × 12 vs. primary ray casting. Although no tessellation artifacts are visible and shading normals are continuous, frame rates with ray casting are even higher on a GeForce RTX 2080 GPU than with rasterization.

On the device side we use a ray generation shader that will generate rays using a pinhole camera model, trace those rays into the scene, and call a user-supplied intersect program when the bounding box of the primitive was hit.

## 4. Results

We compare single-bounce ray casting, path tracing with ten bounces, and rasterization with OpenGL of a massive 280 K glyph data set (see Table 1 and Figure 6). With RTX, we use the robust sphere intersection algorithm from [HGAM19]. With OpenGL, spheres are tessellated to a fixed number of triangles with indexed coordinates and rendered as instances. Even this brute force approach allows us to render a high number of glyphs. Due to instancing, the amount of memory needed is also modest.

We observe that the performance of the OpenGL renderer is dominated by vertex processing load. Especially on the RTX 2080, the results also depend on the field of view, to an extent that suggests that some kind of culling happens at the driver level. When ray tracing, the field of view has an important but inverse effect, which is also more pronounced on the RTX 2080: frame rate drops with the amount of rays hitting objects, even more so with the complexity increased by path tracing. Except when path tracing, frame rates are always interactive. But the OpenGL renderer cannot keep up with tracing primary rays when a tessellation level (24 × 12 vertices) is selected that is sufficiently high to provide for non-objectionable tessellation even when zoomed in.

## 5. Conclusions and future work

Glyph rendering is another domain of scientific visualization where ray tracing hardware acceleration is beneficial, when compared to scanline methods: not only for its improved performance, but also for its increased visual fidelity. At the same time, it opens up a wealth of additional rendering opportunities.

In the future, we aim to combine motion blur with other glyphs than spheres, so that velocity can be shown together with another quantity. We also want to enable evaluation of the effectiveness and usefulness of the proposed methods by making them available to a broader audience by integrating them into production visualization systems such as ParaView or Vistle [Aum19].

## References

- [AGL05] AHRENS J., GEVECI B., LAW C.: Paraview: An end-user tool for large data visualization. In *The Visualization Handbook*, Hansen C. D., Johnson C. R., (Eds.). Academic Press / Elsevier, 2005. 2
- [Aum19] AUMÜLLER M.: Hybrid Remote Visualization in Immersive Virtual Environments with Vistle. In *Eurographics Symposium on Parallel Graphics and Visualization* (2019), Childs H., Frey S., (Eds.), The Eurographics Association. doi:10.2312/pgv.20191113. 4
- [CBW\*12] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., PUGMIRE D., BIAGAS K., MILLER M., HARRISON C., WEBER G. H., KRISHNAN H., FOGAL T., SANDERSON A., GARTH C., BETHEL E. W., CAMP D., RÜBEL O., DURANT M., FAVRE J. M., NAVRÁTIL P.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Oct 2012, pp. 357–372. 2
- [CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. *SIGGRAPH Comput. Graph.* 18, 3 (Jan. 1984), 137–145. URL: <https://doi.org/10.1145/964965.808590>, doi:10.1145/964965.808590. 3
- [GM19] GANTER D., MANZKE M.: An Analysis of Region Clustered BVH Volume Rendering on GPU. *Computer Graphics Forum* (2019). doi:10.1111/cgfm.13756. 2
- [GRE09] GROTTTEL S., REINA G., ERTL T.: Optimized data transfer for time-dependent, GPU-based glyphs. In *Proceedings of IEEE Pacific Visualization Symposium 2009* (2009). URL: <https://doi.org/10.1109/PACIFICVIS.2009.4906839>. 2
- [GRT16] GERRITS T., ROSSL C., THEISEL H.: Glyphs for General Second-Order 2D and 3D Tensors. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2016), 980–989. doi:10.1109/tvcg.2016.2598998. 2
- [HGAM19] HAINES E., GÜNTHER J., AKENINE-MÖLLER T.: *Precision Improvements for Ray/Sphere Intersection*. Apress, Berkeley, CA, 2019, pp. 87–94. URL: [https://doi.org/10.1007/978-1-4842-4427-2\\_7](https://doi.org/10.1007/978-1-4842-4427-2_7), doi:10.1007/978-1-4842-4427-2\_7. 4
- [Kin04] KINDLMANN G.: Superquadric tensor glyphs. In *Proceedings of the Sixth Joint Eurographics - IEEE TCVG Conference on Visualization* (Goslar, DEU, 2004), VISSYM'04, Eurographics Association, p. 147–154. 2
- [MHH\*12] MCAULEY S., HILL S., HOFFMAN N., GOTANDA Y., SMITS B., BURLEY B., MARTINEZ A.: Practical physically-based shading in film and game production. In *ACM SIGGRAPH 2012 Courses* (New York, NY, USA, 2012), SIGGRAPH '12, Association for Computing Machinery. 2
- [Mic18] MICROSOFT C.: Announcing Microsoft DirectX Raytracing!, 2018. URL: <https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/>. 1
- [MRZH14] MÜLLER H., REIHS R., ZATLOUKAL K., HOLZINGER A.: Analysis of biomedical data with multilevel glyphs. In *BMC Bioinformatics* (2014). 2
- [MUWP19] MORRICAL N., USHER W., WALD I., PASCUCCI V.: Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In *2019 IEEE Visualization Conference (VIS)* (Oct 2019), pp. 256–260. doi:10.1109/VISUAL.2019.8933539. 2
- [NVI18] NVIDIA C.: Introduction to Real-Time Ray Tracing with Vulkan, 2018. URL: <https://devblogs.nvidia.com/vulkan-raytracing/>. 1
- [PBD\*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4 (July 2010), 66:1–66:13. URL: <http://doi.acm.org/10.1145/1778765.1778803>. 1, 2
- [PPK17] PODLOZHNYUK A., PIRKER S., KLOSS C.: Efficient implementation of superquadric particles in discrete element method within an open-source framework. *Computational Particle Mechanics* 4, 1 (9 2017), 101–118. doi:10.1007/s40571-016-0131-6. 2
- [RGE19] REINA G., GRALKA P., ERTL T.: A decade of particle-based scientific visualization. *The European Physical Journal (Special Topics)* 227: Particle Methods in Natural Science and Engineering, 14 (2019), 1705–1723. URL: <https://doi.org/10.1140/epjst/e2019-800172-4>, doi:10.1140/epjst/e2019-800172-4. 2
- [ROP11] ROPINSKI T., OELTZE S., PREIM B.: Survey of glyph-based visualization techniques for spatial multivariate medical data. *Comput. Graph.* 35, 2 (2011), 392–401. 2
- [RSMS\*07] ROPINSKI T., SPECHT M., MEYER-SPRADOW J., HINRICHS K., PREIM B.: Surface glyphs for visualizing multimodal volume data. In *Vision, Modeling, and Visualization* (2007), pp. 3–12. 2
- [SK10] SCHULTZ T., KINDLMANN G. L.: Superquadric glyphs for symmetric second-order tensors. *IEEE Transactions on Visualization and Computer Graphics* 16, 6 (Nov/Dec 2010), 1595–1604. 2
- [TL04] TOLEDO R., LÉVY B.: *Extending the graphic pipeline with new GPU-accelerated primitives*. Tech. rep., INRIA-ALICE, 2004. 2
- [Ush19] USHER W.: ChameleonRT, 2019. <https://github.com/Twinklebear/ChameleonRT>. 2
- [WJA\*17] WALD I., JOHNSON G., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GÜNTHER J., NAVRÁTIL P.: OSPRay - a CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan 2017), 931–940. 2
- [WMH20] WALD I., MORRICAL N., HAINES E.: OWL – The Optix 7 Wrapper Library, 2020. URL: <https://github.com/owl-project/owl>. 2
- [WUM\*19] WALD I., USHER W., MORRICAL N., LEDIAEV L., PASCUCCI V.: RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. In *High-Performance Graphics - Short Papers* (2019), Steinberger M., Foley T., (Eds.), The Eurographics Association. doi:10.2312/hpg.20191189. 2
- [WWB\*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph.* 33, 4 (July 2014), 143:1–143:8. URL: <http://doi.acm.org/10.1145/2601097.2601199>, doi:10.1145/2601097.2601199. 2
- [ZHL17] ZELLMANN S., HOEVELS M., LANG U.: Ray traced volume clipping using multi-hit BVH traversal. In *Proceedings of Visualization and Data Analysis (VDA)* (2017), IS&T. 2
- [ZWL17] ZELLMANN S., WICKEROTH D., LANG U.: Visionaray: A cross-platform ray tracing template library. In *Proceedings of the 10th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (IEEE SEARIS 2017)* (2017), IEEE. 2