




Efficient Sphere Rendering Revisited

P. Gralka¹ , G. Reina¹ , and T. Ertl² 

¹University of Stuttgart, Visualization Research Center, Germany

²University of Stuttgart, Institute for Visualization and Interactive Systems, Germany

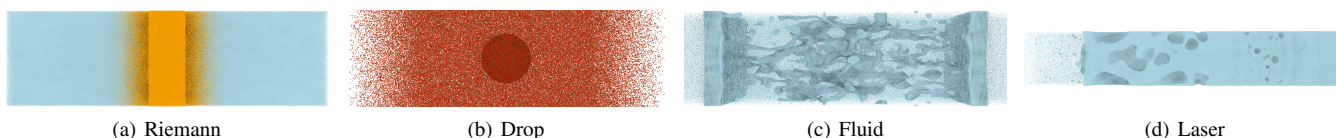


Figure 1: Renderings of the datasets in static scenes used to evaluate the different rendering pipelines and methods. The datasets (a), (b), (c) are results from simulations in the area of process engineering. They typically contain dense filaments or droplets of liquid surrounded by gas with a low particle density. In contrast, the dataset (d) contains a large crystalline bulk of matter with a high density and a comparably low amount of particle clusters released from the bulk by a laser.

Abstract

Glyphs are an intuitive way of displaying the results of atomistic simulations, usually as spheres. Raycasting of camera-aligned billboards is considered the state-of-the-art technique to render large sets of spheres in a rasterization-based pipeline since the approach was first proposed by Gumhold. Over time various acceleration techniques have been proposed, such as the rendering of point primitives as billboards, which are trivial to rasterize and avoid a high workload in the vertex pipeline. Other techniques attempt to optimize data upload and access patterns in shader programs, both relevant aspects for dynamic data. Recent advances in graphics hardware raise the question of whether these optimizations are still valid. We evaluate several rendering and data access scheme combinations on real-world datasets and derive recommendations for efficient rasterization-based sphere rendering.

CCS Concepts

• *Computing methodologies* → *Rasterization*; • *Human-centered computing* → *Scientific visualization*;

1. Introduction

Simulating phenomena in process engineering, biology, or cosmology often produces large systems of particles representing atoms, molecules, or celestial bodies. Typical data sizes range from a few particles up to millions and billions. A suitable metaphor to visualize particles in such data is a sphere. Gumhold [Gum03] presented a method to utilize a rasterization-based rendering pipeline for sphere rendering by ray casting using imposter geometry/billboards. The basic concept behind this method is still considered state-of-the-art. Other rendering methods, such as ray tracing of large systems [WKJ*16], have been proposed since then, as well as revisiting the classical approach employed in fixed-function rasterization, i.e., using explicitly tessellated geometry [ZAMW20]. The inclusion of dedicated ray tracing processors in the architecture of GPUs made the interactive ray tracing of large particle systems feasible on GPUs as well. However, Gralka et al. [GWG*20] show that there are use cases and scenarios for which glyph-based ren-

dering in a rasterization pipeline outperforms ray tracing. The main advantage is the possibility of getting an immediate overview of the data without the requirement of sorting, structuring, and other pre-processing steps. Hardware and rendering APIs, such as OpenGL, evolve over time, including the introduction of new features such as the mesh shader. Drivers change with each version and prioritize different aspects of an API for optimization. In this paper, we revisit OpenGL-based rendering pipelines for sphere rendering as previously presented in the overview by Falk et al. [FGKR16], investigate their performance on current hardware and whether the assumptions made in this overview still hold. Additionally, we investigate how well the mesh shader competes against these “classical” pipelines. In the evaluation, we investigate different input primitive types, composition and parametrization of shader stages. The main focus is on the rendering performance itself. The evaluation does not consider other aspects, such as efficient data upload in OpenGL [Wie21].

© 2023 The Authors.

Proceedings published by Eurographics - The European Association for Computer Graphics.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

2. Related Work

Over time various approaches have been proposed to render spheres, such as splats [RL00], explicit geometry [ZAMW20], or implicit parametrization [Gum03, WKJ*16]. Methods have been proposed for both, rasterization-based pipelines, as well as ray-tracing-based pipelines. Typically, spheres represent datasets containing unstructured data. For ray-tracing-based pipelines, a spatial acceleration data structure [PGSS07, PL10, WKJ*16, GWG*20] is generated for the data during pre-processing to accelerate rendering. In contrast, in rasterization-based pipelines, millions of points can be rendered interactively without pre-processing [GKM*15]. However, since the input data is typically unstructured and unsorted, the number of primitives as well as the overdraw limit rendering performance. Computing a per-fragment correct depth ensures correct occlusion and intersection of the spheres. Hardware features such as the conservative depth extension allow combining the modification of the depth of a fragment together with early z-culling to help alleviate the overdraw issue [MBE19]. Unfortunately, the effectiveness of conservative depth depends on the sorting of the data in direction of the camera. Culling based on occlusion can help reduce the data size required for transfer to the GPU [GRDE10], a common rendering bottleneck [GRE09]. Hierarchical data structures are also used to reduce dataset sizes and to improve the scalability of rendering pipelines [HE03, FSW09]. Other acceleration techniques include level-of-detail rendering [RHI*15] or hierarchical ray casting [FKE13], aiming at reducing rendering quality or data bandwidth for areas in the scene that are expected to have a low contribution to the final image. In certain domains, such as biomolecules, instancing [LBH12, FKE13] of repeating structures further reduces the rendering scene’s size. An accurate occlusion model can improve the rendering quality at a moderate performance cost [IRR*22]. While additional effects that improve the perception of data, such as shadows and transparency, are intuitive to implement in a ray-tracing-based pipeline, these effects can be approximated in a rasterization-based pipeline [GKSE12, SGG15], as well.

Some point-based rendering (PBR) aspects are technically similar to glyph-based rendering. Gross and Pfister [GP07] provide a comprehensive overview of PBR techniques, some of which are the basis of sphere rendering approaches still in use, while Berger et al. [BTS*16] provide a more recent survey. PBR techniques are typically employed to visualize datasets representing surfaces, for instance, a laser scan of a building or object. This property is typically utilized, sometimes in conjunction with sophisticated data structures, to accelerate rendering. While many glyph-based datasets, for instance, molecular dynamics simulation runs, would lose too many details if transformed into a surface representation, some of the acceleration data structures are applicable. However, we focus this investigation on the performance of the final rendering step, explicitly excluding any pre-processing. Therefore, we refrain from an in-depth analysis of PBR techniques. Lessons learned can still be relevant to PBR techniques as some rely on a similar final rendering step, with the main difference being what is sampled in the fragment stage, for instance, an implicit sphere or a Gaussian kernel splat.

Many of the above-mentioned rasterization-based techniques have the ray casting method introduced by Gumhold [Gum03] or a method derived from that as a central building block. This method

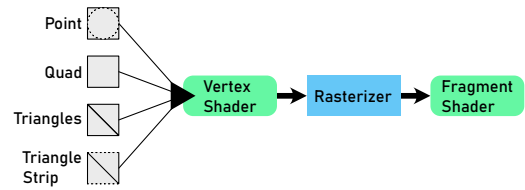


Figure 2: Standard OpenGL rendering pipeline. All standard pipelines that we investigate use at least the vertex stage and the fragment stage. Only the tessellation shader stages are left out. We do not see a benefit for the use case of billboard ray casting. The tested standard pipelines differ in the input geometry type.

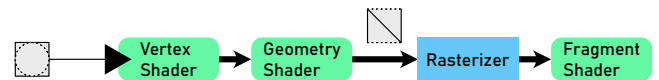


Figure 3: Standard OpenGL pipeline with geometry shader. This pipeline is launched with the point primitive. It is transformed into a triangle strip in the geometry shader.

is subject to optimization and evaluation [GRE09, FGKR16] in line with available hardware features and rendering APIs. Hardware and APIs evolve over time. A new feature that could potentially improve the performance of this basic building block, and therefore all derived methods, is the mesh shader stage, which has already been used by Ibrahim et al. [IRR*22] on pre-processed clusters of particles. Therefore, the subsequent investigation compares different rendering pipelines and billboard primitives for facilitating the ray casting method of Gumhold [Gum03].

3. Overview

There are many ways to render spheres in the context of scientific datasets. We look into “classic” rasterization-based methods, the novel mesh shader, and ray tracing. We understand “classic” methods as approaches utilizing the rasterization pipeline provided by APIs such as OpenGL either with explicit sphere geometry or with billboards enabling ray casting in the fragment shader. Rendering pipelines using explicit geometry have recently been evaluated and compared with ray tracing by Zellmann et al. [ZAMW20] in terms of performance and rendering quality. Therefore, we focus on billboard rendering that allows to implement ray casting on a rasterization-based rendering pipeline, a rendering technique in line with Gumhold [Gum03]. The basic idea is to produce a planar billboard, for instance, a point primitive or a quad, for each sphere. Each primitive covers the entire (projected) extent of each sphere. The rasterizer separates each billboard into a set of fragments. For each of these fragments, a ray can be generated. Since the billboard covers the entire extent of a sphere, we can guarantee that the entire sphere is sampled with these rays. Occlusion is solved using the depth buffer of the rasterization-based pipeline. Over time others have proposed extensions of this basic idea to introduce approximations of global effects, such as shadows [GKSE12] or transparency [SGG15], or acceleration methods, such as occlusion culling [GRDE10], to reduce the workload in the pipeline. These pipelines require at least

the vertex and fragment shader stage and sometimes incorporate the geometry and tessellation shader as well (see Figure 2 and Figure 3).

Newer generations of GPUs provide with the mesh shader a novel functionality that combines all programmable vertex processing stages into one program (see Figure 6). Although intended to reduce the recomputation of redundant data in complex meshes, a mesh shader stage could potentially reduce overhead for processing billboards, as well, especially if polygonal billboards have to be used, in the absence of (large enough) point primitives.

Finally, ray tracing gained traction in scientific contexts with frameworks such as OSPRay [WJA*17] and OptiX [PBD*10]. Especially on hardware with specific accelerators like modern GPUs, sphere ray tracing achieves high performance and a high rendering quality through global lighting effects. However, unlike the “classic” rasterization-based methods that can typically render data as they are, a ray tracer requires the data to be reordered into an acceleration data structure that, in most cases, adds significant overhead to the data, typically a factor of 3 for a bounding volume hierarchy. This can be prohibitive for use cases with especially large data.

4. Implementation

In the following, we present details on the implementation of the different rendering pipelines. These pipelines are embedded in the visualization software MegaMol [GBB*19]. The pipelines rely on the provided facilities for data ingestion, pre-processing, compositing, and display of framebuffer contents, which could impact absolute performance. However, we measure performance at the draw call level using OpenGL timer queries that exclude all the software-specific overhead. This minimizes the impact of the used software environment on the results and allows for the generalization of our findings. The code can be found on GitHub [Meg23] (*SRTTest.cpp*).

The “classic” billboard rendering pipelines and their variations are extensively described in [FGKR16]. We refer the reader to this book for details and only describe the basics necessary to understand the evaluation and discussion, as well as changes and observations concerning methods described in the book. We describe the mesh shader pipeline in more detail. We use the code provided alongside [GWG*20] to provide baselines for comparison with ray tracing.

4.1. Data Access

The first step in every rendering pipeline for sphere ray casting is fetching input data. Since we are considering rendering on dedicated GPUs, moving the data into VRAM plays an important role. However, for simplicity, we assume the data is already available statically in VRAM and focus on evaluating data layouts and access patterns in the shader stages. Several recent publications have evaluated efficient upload methods in modern rendering APIs [FGKR16, WK19, Wie21].

As mentioned in Section 3, the advantage of rasterization-based methods compared to ray tracing is that no costly re-ordering of data is required. Contents of a dataset can directly be streamed onto the GPU for rendering. However, this requires flexible and adaptable data transfer to and data access in shader stages.

One way to provide the shader stages with input is a Vertex Array Object (VAO). A VAO can transport a set of attributes that define the type and size of data streams, such as a set of `vec3` for particle positions. In modern OpenGL, a buffer object, such as a vertex buffer, is bound as backing storage to an attribute. However, available attribute layouts are restricted.

Shader Storage Buffer Objects (SSBO) provide more flexibility and are capable of transporting arbitrary data structs. They have mainly two essential constraints: The maximum contiguous memory size is vendor-dependent (usually 2 GB), and the alignment of the data within the SSBO stored as array-of-structs (AoS) is constrained to pre-defined standards in OpenGL, such as `std430`. Alternatively, data can be transported as struct-of-arrays (SoA) by multiple SSBOs. We explore three different layouts:

AoS. compact and fully interleaved layout consisting of `vec4`

partial SoA. separation of position and color like in the VAO

SoA. separate buffer for each coordinate (stream of `float` values).

The first data layout represents the concept of an array-of-structs that could have a better performance due to a potentially smaller footprint in the cache. The second and third data layouts are struct-of-arrays representatives. However, the latter layout is more in line with the internal data representation of the visualization software MegaMol [GBB*19], as well as the internal data representation of some simulation code (for performance reasons [FSS13]), allowing transfer of data from the simulation directly to the visualization without the need of transformation. Such a layout also resembles that of columns of tabular data allowing a cheap and straightforward slicing of a larger simulation dataset into smaller visualization datasets, reducing upload bandwidth requirements, which is a typical bottleneck of GPU rendering.

To avoid copying data in host code to assemble it into a pre-defined data layout for upload to GPU and access in the shader stages, the shader code requires adaptable accessors for arbitrary input data. In the host code domain, typically C++, such flexibility is provided through generics or virtual function calls. Both features are not available in standard OpenGL shader language. Falk et al. [FGKR16] suggest the injection of custom code snippets into the code of the respective shader stages before the compilation by OpenGL.

However, such a feature can also be mimicked through a non-standard pre-processor able to parse `#include` directives and exploit fast recompilation of shader code. If the shader is required to access buffers with differing data layouts, we can provide a common method declaration, such as `access_data(idx)`, called within the main function of a shader stage, while the dataset-dependent implementation is hidden in different include files that can be exposed on-demand by the pre-processor, thus emulating virtual function calls within shader code. We use a custom shader pre-processor based on `glslang` [Khr21] to switch between features regarding data layout, proxy geometry types, etc.

4.2. Billboard Ray Casting

This investigation is focused on implicit sphere representations, but the following methods can be generalized for any simplex with an implicit definition. While in image-order rendering methods, such as

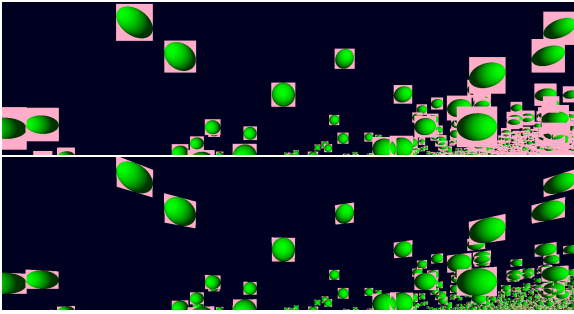


Figure 4: *Difference between screen and camera orientation of the billboards. Top: Screen-aligned point primitive billboards. Bottom: Camera-aligned triangulated billboards. Camera-aligned billboards produce less excess fragments caused by the perspective distortion.*

ray tracing, implicit geometry can be directly sampled, object-order rendering methods, such as rasterization-based pipelines, require some explicit geometry to facilitate rendering. In our case, these are individual planar billboards covering the projected extent of an implicit sphere. The billboards’ geometry can either be a point primitive or a triangulated polygon. The orientation of the point primitive billboards is by design aligned with the screen. In the case of a perspective camera setup, the polygon billboard can be aligned toward the camera. Looking at the different orientations while refraining from discarding excess fragments, as shown in Figure 4, camera-aligned billboards produce fewer fragments, especially for spheres at the viewport’s edges and in the case of a large field-of-view. Since a ray is evaluated for each fragment, potentially fewer intersection tests are computed as a consequence. The performance impact is discussed later.

4.2.1. Classic Rendering Pipeline

We understand as a “classic” rendering pipeline for billboard ray casting what can be seen in Figure 2—a basic pipeline consisting of a vertex shader stage and a fragment shader stage. The underlying algorithm is presented in detail in [FGKR16]. The important step is determining the billboard’s extent in screen space. It produces four sample vertices—touch points of tangents on the sphere surface with the origin in the camera position—to determine the billboard’s size and the center. The calculations can be conducted in 2D by projecting the camera-object rays onto the xz - and yz -plane, simplifying the computations. These computations are conducted in the vertex shader. The algorithm uses point primitives as input geometry. Even if the spheres have a fixed radius, due to projection, the rendered points have varying sizes that require the pipeline to support per-vertex point sizes. There are vendor-specific limits to the maximal admissible point size, and typically the limit is not sufficient for close-up scenes such that spheres become larger than their billboard and are cut at their outline. However, other geometry types are possible that are not impeded by this issue, such as quads, triangles, and triangle strips. In these cases, the billboard is not a point but a polygon, and the vertex shader computes the positions of the corner vertices. We modified the algorithm in [FGKR16] accordingly for screen-aligned polygonal billboards.

Alternatively, one can use the algorithm for computing the corner vertices shown in Listing 1 and Figure 5. It results in a camera-aligned billboard, which works only with polygonal billboards, i.e., a plane in the sphere center perpendicular to the direction between the camera and the sphere center. The corner vertices of the billboard are the intersections of the sphere tangents with the plane. This method is based on the observation that the opening angle α at the camera is the same angle at the sphere center M . Since $vi = r / \cos \alpha$ we can utilize this equality through $\cos^2 = 1 - \sin^2$ and $\sin \alpha = r / CM$.

Another option to avoid the restrictions of point primitives is a geometry-shader-based pipeline, as shown in Figure 3. The purpose of a geometry shader stage is to produce additional geometry within the rendering pipeline. The proposed pipeline is initiated with point primitives and converts the points in the geometry shader stage to a triangle strip. The advantage is that all four vertices of the triangle strip are created in a single geometry shader instance. Therefore, the computation of their positions is also conducted in one invocation. In the vertex shader stage, every invocation would repeat these computations. More importantly, since no cooperative computation model is available in this shader stage, each invocation would fetch the same sphere position to initiate the computation. We refrained from using the tessellation shader since it would computationally mimic the geometry shader at the cost of an additional shader stage.

A common aspect of all these pipelines is that there is no backing input geometry computed in host code or loaded from a file for the pipeline launches. Every vertex is generated on-the-fly in the pipeline. We only have to ensure that enough vertex shader instances are spawned. Therefore, we refrained, in the general case, from using instancing as it does not reduce the number of required vertex shader instances nor reduces the number of draw calls. However, we observed a significant performance gain from instancing in the triangle strip case on NVIDIA GPUs. All other pipelines are launched with a simple `glDrawArrays` call.

Acceleration of the rendering, especially for vertex-heavy geometry types such as triangles, could be achieved by early culling of spheres against the frustum in the vertex processing stage of the rendering pipeline. In the case of billboards, the culling is determined by the vertices of the billboard—if all vertices are outside the frustum, it is safe to assume that the complete sphere is outside, as well. There are two ways to enforce this culling. In a standard OpenGL rendering pipeline, the per-vertex output parameters `gl_ClipDistance` and, in later API versions, `gl_CullDistance` are available to support user-culling. A negative clip or cull distance determines the respective vertex to be outside the frustum. In contrast to the clip distance, a single negative cull distance ensures the removal of the entire primitive. This makes the cull distance useful for the use case of sphere rendering because the invocations of the vertex shader are independent of each other. Therefore, it would be necessary to redundantly compute the clip distance for all vertices in every shader invocation to ensure a coherent culling of a billboard outside the frustum. However, activating user-culling adds an overhead to the execution of a rendering pipeline such that we have not found a use case with an overall performance gain. Therefore, we discarded this concept in the implementation of the rendering pipelines.

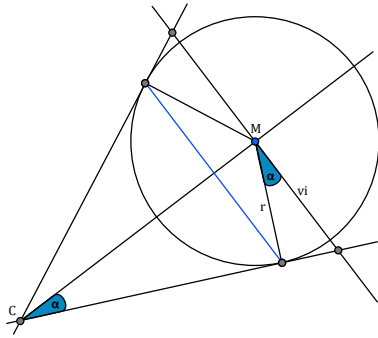


Figure 5: Geometrical representation of the algorithm computing a camera-aligned billboard in 2D. The opening angle α derived from the field-of-view is the same angle at the sphere center M . This allows to find the minimal width of the billboard required to cover the entire projected extent of the sphere.

4.3. Mesh Shader

The billboard construction methods mentioned above can be mapped to the new mesh shader stage. The concept of mesh shaders [NVI22] aims at introducing the cooperative execution model of compute shaders into the vertex processing of a rendering pipeline. This concept tries to optimize on-chip memory usage to run as many tasks in parallel as possible. A task is defined in terms of a so-called meshlet. The term refers to a subset of a large mesh. However, since the input and output of a mesh shader are almost entirely user-defined, as well as the inner representation of a meshlet, it could also refer to a set of particles or any other type of geometry. The mesh shader stage replaces all vertex processing stages of a “classic” OpenGL pipeline, including vertex, geometry, and tessellation shader stages, reducing the fixed-function impact on the pipeline. In an invocation of a mesh shader, a set of vertex positions and attributes are determined, and a set of indices emitted that defines the topology of the vertices, thus combining the functionality of the vertex and geometry stage with the flexibility of arbitrary input and output. This allows for re-using vertices in complex mesh topologies. Since the computational model follows that of compute shaders, shared memory and other collaborative functionality, such as balloting, can be utilized by mesh shaders. The main performance aspect is controlled by the size of the output data of the shader stage and the amount of accessed data. Output allocation is done in batches of 128 bytes. To optimize performance, output in terms of the number of vertices, number of primitives, and size of per-vertex attributes should be minimal and aligned with that size (which includes the final 4 byte primitive count).

The mesh shader pipeline we use is shown in Figure 6. In our case, the mesh stage either emits a set of point primitives or pairs of triangles. Since we can emit vertices and indices, we can reduce the output per billboard to four vertices and six indices, forming two triangles compared to the six vertices we require in the “classic” rendering pipeline. Additionally, similar to the geometry shader case, the computation of the corner vertices is conducted once per billboard and is not repeated for every vertex. The input is the same for both the point and triangle case, i.e., 31 spheres per input meshlet,

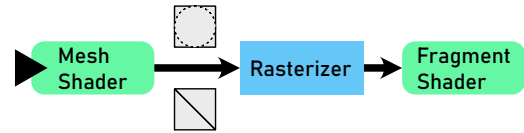


Figure 6: A general mesh shader pipeline. It reduces almost all programmable and fixed-function stages into a single shader stage. Input and output are similar to a geometry shader stage. The available functionality is similar to a compute shader stage. We test the mesh shader pipeline with either points or triangles as output.

which is also the local workgroup size that has to be defined similarly to compute shaders. The maximal output size and layout, in terms of vertices, primitives, as well as primitive type, is pre-defined, similar to geometry shaders. Listing 2 shows the code for the point-emitting mesh shader.

Additionally, one can prepend a task shader before a mesh shader, which can further improve performance. This shader stage is used to emit mesh shader tasks. Therefore, the task shader is useful for early culling of geometry or selection of subdivision levels. However, utilizing the task shader adds overhead to the pipeline that could thwart performance gains, which is the case for our purposes.

5. Results

We evaluate the different rendering pipelines with four test datasets. Most of the real-world examples are from process engineering. These datasets typically feature a distinct variation in particle density as they contain filaments and droplets of dense fluid embedded in sparse gas, such as *Fluid* (see Figure 1(c)) and *Drop* [EV15] (see Figure 1(b)). *Riemann* [HHVM20] contains a large dense fluid block in the middle of the dataset surrounded by gas similar to *Drop* but with an overall higher density (see Figure 1(a)). *Laser* [ETR19] dataset (see Figure 1(d)) contains a large and dense metal crystal shot with a high-energy laser that released several larger and smaller clusters of particles from the crystal. Most of the bounding box is filled with a dense crystal. General statistics of the datasets are shown in Table 1. Additional results can be found in the supplemental material.

Table 1: Properties of the test datasets.

	Riemann	Drop	Fluid	Laser
#parts	306,112,864	3,999,672	29,999,997	199,885,091
size	4.898 GB	0.064 GB	0.480 GB	3.198 GB
radius	0.5	0.0008	0.5	2.023

5.1. Tested Methods

The test setup focuses on the primitive type and used shader stages to reveal performance differences on modern hardware and scalability considering dataset size. For conciseness, the evaluation is limited to rendering performance, excluding data pre-processing (for instance, acceleration data structure construction) and data upload (assuming static availability of all data, including time-dependent datasets). We look at the “classic” rendering pipeline consisting of a vertex

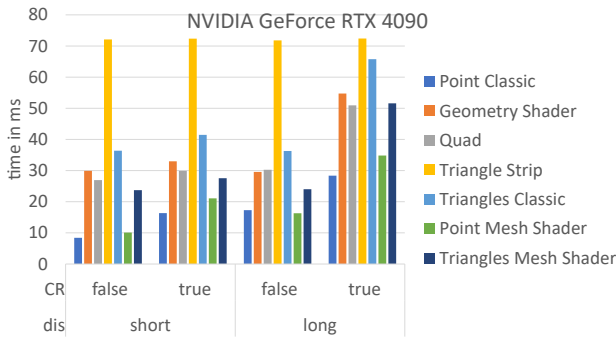


Figure 7: Rendering time results in milliseconds on the NVIDIA RTX 4090. The results are averaged over all four datasets. The label *dis* refers to the distance of the camera orbit to the dataset. The label *CR* indicates whether the conservative rasterization extension is enabled or not.

and fragment shader stage with the point, quad, triangle, and triangle strip primitives. Another test case adds a geometry shader stage. In this pipeline points as base input is converted into triangle strips in the geometry shader stage. Finally, we test a mesh shader pipeline that emits either points or triangles.

For the tests, the datasets are rendered from ten different camera configurations sampled around an orbit. To evaluate the impact of point sizes on the different rendering pipelines and primitives, we test two orbits at different distances (short—within the dataset bounding box; long—outside the dataset bounding box). The median rendering time in milliseconds of every recorded frame is reported. The results are aggregated over all camera scenes and all datasets to evaluate and discuss trends and relative performance between the methods rather than dataset-specific issues.

If nothing else is stated, the pipelines use screen-aligned billboards, a `vec4` stream as input, and have conservative depth enabled. The machines used to measure the performance of the pipelines (*c* indicates hardware cores, SMT/HT is active on all machines):

1. AMD Ryzen 7 3800X (8 c @ 3.9 GHz); 96 GB DDR4 RAM (@ 1,600 MHz); NVIDIA Geforce RTX 3090 (512.77) 24 GB
2. AMD Ryzen 9 5900X (12 c @ 4.7 GHz); 64 GB DDR4 RAM (@ 1,333 MHz); AMD Radeon PRO W6800 (22.Q4) 32 GB
3. AMD Ryzen 9 5900X (12 c @ 4.7 GHz); 64 GB DDR4 RAM (@ 1,333 MHz); AMD Radeon RX 6900 XT (22.Q4) 16 GB
4. AMD Ryzen 9 5900X (12 c @ 4.7 GHz); 64 GB DDR4 RAM (@ 1,333 MHz); NVIDIA Geforce RTX 3090 Ti (531.18) 24 GB
5. AMD Ryzen 9 5900X (12 c @ 4.7 GHz); 64 GB DDR4 RAM (@ 1,333 MHz); NVIDIA Geforce RTX 4090 (531.18) 24 GB
6. AMD Ryzen 9 5900X (12 c @ 4.7 GHz); 64 GB DDR4 RAM (@ 1,333 MHz); Intel Arc A770 (31.0.101.4146) 16 GB

We use query objects in the OpenGL implementations. A query is declared by `glQueryCounter` to record a timestamp. A query object ensures that all OpenGL commands issued before the query declaration are realized. The results of the queries are collected later once the OpenGL server signals their availability. Synchronizing the state machine explicitly is not necessary if using query objects.

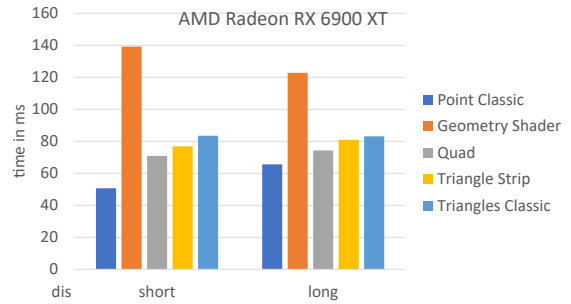


Figure 8: Rendering time results in milliseconds on the AMD Radeon RX 6900XT. The results are averaged over all four datasets. The label *dis* refers to the distance of the camera orbit to the dataset.

We limit the measured region to the draw commands to exclude any potential influence on the results from the visualization software in which the pipelines are implemented. This especially consists of compositing and framebuffer presentation, as well as general overhead for visualization pipeline updates between frames. The potential impact of input data layout that is dictated by the software is discussed in Section 5.3 to derive generalizable results independent of the underlying visualization software.

5.2. Rendering Times

We compare the raw rendering times in Figure 7, Figure 8, and Figure 9 (underlying data [GRE23]). The results show that the point primitive performs best on all tested systems except on the Intel A770 GPU. Intel officially states that the drivers at the time of writing focus on modern APIs, such as Vulkan, so this is expected.

For the other vendors, there is a significant performance gap between the point primitive and the pipelines rendering polygonal billboards, including the geometry shader pipeline. Ignoring the outliers—triangle strip on NVIDIA GPUs and geometry shader on AMD GPUs—the gap between the point primitive and the worst polygonal candidate is in the range of roughly 110 % to 332 % and 27 % to 65 % respectively, depending on the rendering distance. As the lower gap on AMD hardware indicates, the variation between the methods is smoother, and almost linear. Note that the measurements were taken without the stable power state setting and locked clocks to be closer to real-life usage.

Changing the rendering distance from short to long has a higher impact on the point primitive pipelines. We assume this is because many small triangles are culled in the pipeline while points smaller than one pixel are rasterized to one pixel. Activating conservative rasterization—an NVIDIA-specific extension—the long rendering distance using the polygonal billboards is affected, indicating that this extension might be required for a density-preserving rendering with polygonal billboards.

In the case that the point primitive cannot be used due to the size limits, the quad geometry provides the lowest rendering time on both NVIDIA and AMD GPUs. However, `GL_QUADS` is deprecated and requires a forward-compatible rendering context. The performance for the remaining polygonal billboard pipelines is not consistent

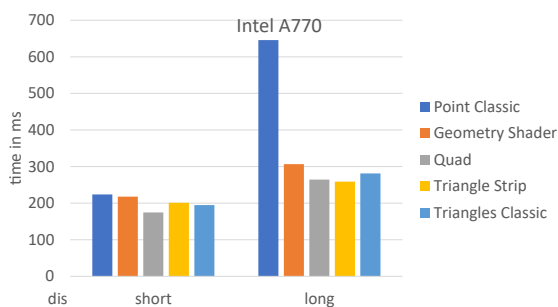


Figure 9: Rendering time results in milliseconds on the Intel A770. The results are averaged over all four datasets. The label dis refers to the distance of the camera orbit to the dataset.

between the vendors. On the one hand, the geometry shader performs well on NVIDIA GPUs but renders significantly slower compared to the polygonal counterparts on AMD GPUs. On the other hand, the triangle strip geometry is reasonably fast on AMD GPUs, while it results in the worst performance on NVIDIA GPUs, almost as if the rendering time has a lower bound for this geometry.

If the mesh shader feature is available, it is the best choice for polygonal billboards. There is no significant performance gain using this shader feature for point primitives. The simple scaling benchmark shown in Figure 11 confirms this observation. We use randomly distributed particles and increase their number. The point primitive pipeline performs best, with the mesh-shader-based pipelines having roughly similar performance, slightly below the classic point-based pipeline but still better than any other variant. The AMD platform shows a similar scaling behavior excluding the respective outliers geometry shader pipeline and triangle strip primitive.

The results in Figure 7, Figure 8, and Figure 9 are averaged over all datasets. To preclude dataset-specific effects on the results, we plot the rendering time variance in Figure 10. The overall ranking of the methods persists, as shown in Figure 7. Additionally, the evaluation of the variance reveals that the point-primitive-based pipelines have a more stable performance across the different datasets. Again, the AMD platform shows similar behavior.

Table 2 shows baselines for static scenes rendered with techniques that require pre-processing. The ray tracer is the implementation from [GWG*20] based on OptiX [PBD*10]. The datasets are rendered with the built-in bounding volume hierarchy as an acceleration structure that benefits from both the hardware-accelerated traversal on the GPU, and the Pkd-tree [WKJ*16]. The BVH provides the best overall results but runs into memory issues on system 1 due to the high memory overhead of the data structure. A Pkd-tree is specifically designed to encode the hierarchy of the acceleration data structure in place, showing that pre-processing of data, in use cases that allow that, can improve rendering performance over the “brute-force” pipelines evaluated in Figure 7, Figure 8, and Figure 9. However, it struggles with the *Drop* dataset due to how the hierarchical structure handles the dense droplet in the middle of the dataset. The occlusion culling renderer [GRDE10] constructs a grid over the dataset and first issues coarse occlusion queries per cell and

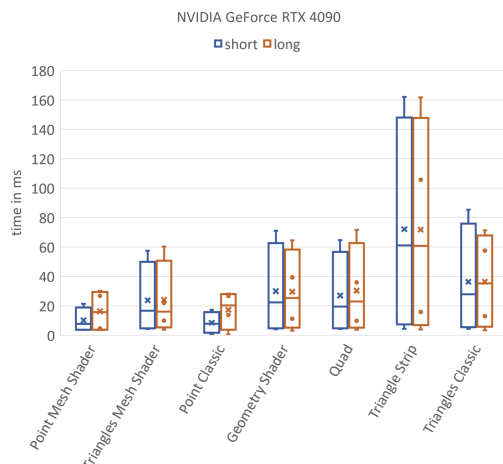


Figure 10: Rendering time results in milliseconds on the NVIDIA RTX 4090. The results show the variance between the different test datasets. The point primitive pipelines have a lower variance and therefore perform more stable across datasets.

Table 2: Frame times in milliseconds reported for ray tracing (BVH, Pkd), occlusion culling (OC), and the point primitive in a “classic” rendering pipeline. The datasets are rendered on system 1 in static scenes as shown in Figure 1 as baselines for comparison. (OOM: out of memory)

	Riemann	Drop	Fluid	Laser
Point	75.1	1.3	10.4	70.1
BVH	OOM	1.36	1.07	OOM
Pkd	15.21	18.26	8.45	5.68
OC	14	18	>1000	11

later per glyph. As expected, this culling technique performs well compared to the “brute-force” pipelines with dense datasets, such as *Riemann* and *Laser*, with many occluders. It struggles with the sparse datasets *Drop* and *Fluid* to improve performance over the “brute-force” approaches.

5.3. Data Access

We evaluated different data layouts of the SSBOs used to store the dataset. The scaling results shown in Figure 12 reveal no relevant impact on rendering performance based on layout but instead based on overall size. There are two distinct groups (within a 1 ms range). One group comprises layouts representing eight fully separated float streams and two vec4 streams for position (+ radius) and color. The other group comprises the interleaved layout (AoS with vec3 position and uint color) and a single vec3 position stream. There is also no noticeable difference between the input layouts on the AMD platform. Since the layout has no significant impact on runtime, the results in Section 5.2 should apply to other visualization software utilizing the OpenGL API.

Different layouts can be encapsulated as described in Section 4.1. This argues in favor of adopting the data layout for rendering in

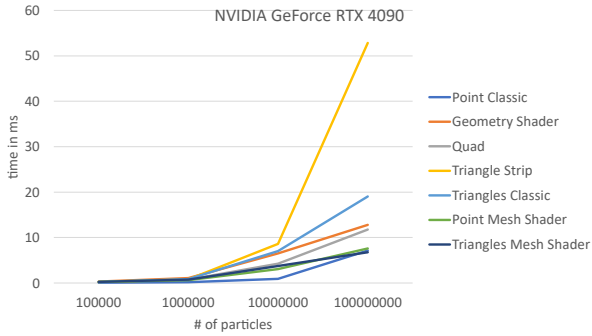


Figure 11: Rendering time results on the NVIDIA RTX 4090. Synthetic datasets are used with the respective number of randomly sampled particles.

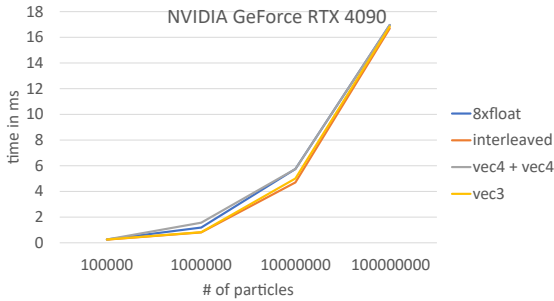


Figure 12: Rendering time results in milliseconds on the NVIDIA RTX 4090. We compare different input data layouts. Synthetic datasets are used with the respective number of randomly sampled particles. Input size seems to have more influence than layout.

line with the internal data representation of the source, such as simulation software like ls1 [NBB*14]. This particular software separates each attribute and each coordinate into a separate `float` stream. Besides being able to directly map the data into the rendering pipeline, keeping this layout has the potential to improve the rendering performance, as it allows the pipeline to better take advantage of the vendor-specific size limitations of SSBOs, such that more particles can be rendered in a single draw call.

5.4. Screen-aligned vs. Camera-aligned Billboards

The results reported in Section 5.2, especially with enabled conservative rasterization, indicate that the overall fragment processing has a high impact on performance and could be a potential focus of optimization. As shown in Figure 4, camera-aligned billboards can reduce the excess fragments compared to screen-aligned billboards as they better fit the sphere outline in perspective camera scenes. We evaluated both alignments in Table 3 with enabled conservative rasterization since we expect the highest fragment workload with this extension. However, we observe only a marginal improvement over screen-aligned billboards. Looking at the number of pixel shader launches, the camera-aligned billboards fail their purpose as the number increases. Only at extreme aperture angles, such as 140° ,

Table 3: Comparison between screen-aligned and camera-aligned triangulated billboards with enabled conservative rasterization. The datasets are rendered on system 1 with multiple camera configuration sampled on an orbit around the data at distance (*dis*) relative to the longest bounding box edge.

Type	align	dis	Riemann	Drop	Fluid	Laser
Quad	screen cam	l	293.08	2.86	31.49	267.76
			292.93	1.72	32.02	268.37
Mesh	screen cam	l	295.08	3.3	32.78	272.47
			302.48	1.76	32.68	272.99
Quad	screen cam	s	179.48	2.63	17.27	165.45
			177.08	1.36	17.13	163.9
Mesh	screen cam	s	181.14	2.75	17.9	168.88
			180.15	1.42	17.98	168.27

Table 4: Comparison of two different input meshlet sizes for the mesh shader stage. We compare point-based (*point*) and triangulated (*Tri*) billboards without conservative rasterization. The datasets are rendered on system 1 with multiple camera configuration sampled on an orbit around the data at distance (*dis*) relative to the longest bounding box edge. Rendering times are reported in milliseconds.

Type	dis	Size	Riemann	Drop	Fluid	Laser
Tri	l	31	36.67	3.44	4.0	33.17
		15	36.42	1.23	3.9	33.81
Tri	s	31	48.12	3.61	8.34	90.7
		15	47.22	1.21	8.15	90.56
Point	l	31	101.22	0.89	11.35	89.34
		15	104.83	0.68	11.5	92.4
Point	s	31	59.28	1.45	5.77	48.47
		15	60.64	0.87	6.02	51.01

the camera-aligned billboards result in fewer pixel shader launches. However, the difference is still marginal, between 1 and 2% in the observed cases.

5.5. Meshlet Size

One user-adjustable parameter of mesh shaders is the size of an input meshlet in terms of the entities it contains. This choice influences the size of the output per mesh shader invocation. The larger the output allocation the fewer invocations can be processed in parallel. Therefore, we evaluated two different input particle counts for our mesh shader pipelines in Table 4. We observe no difference outside a typical error margin for our measurements.

5.6. Performance Counters

To find possible explanations for the relative performance differences the overview of the rendering times revealed, we drill down to the details of the resource utilization on the hardware level. Vendor-specific performance counters provide these details. We chose NVIDIA GPUs as the test platform, which are currently the only rendering hardware to expose mesh shaders in OpenGL. The NVPerf library [NVI23] allows us to instrument the rendering code,

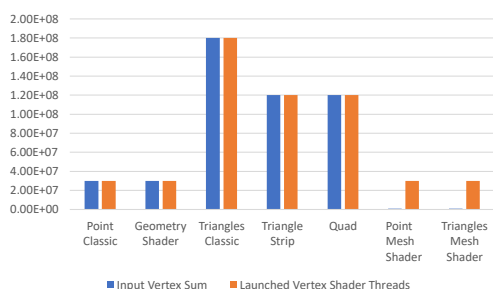


Figure 13: The overall sum of vertices as input for each tested pipeline and number of launched vertex shader threads. Triangles result, with six vertices per billboard, in the highest vertex input, while both the point and geometry shader pipelines are initiated with point primitives requiring only one vertex per billboard. For each of these vertices, a vertex shader thread is launched. Results recorded on system 1.

automatically producing a comprehensive record of all available performance counters and their values within specified test regions. The region we use is limited to the draw calls for the tested method. The library automatically renders a specific number of frames and aggregates the collected metrics per test run.

Some of the results fit expectations and can serve for verification: The results on the primitive distributor (see Figure 13) that executes before the shaders has the highest vertex count for the `GL_TRIANGLES` primitive with six input vertices per billboard, followed by `GL_TRIANGLE_STRIP` and `GL_QUADS` with four, and finally the `GL_POINTS` primitive with one. No input vertex count is recorded for the mesh shader types, probably because a different hardware unit is responsible for input distribution. The input vertex count directly translates to the number of launched vertex shader threads (see Figure 13). We cannot observe a similar direct relation between the launch of the fragment shader and rasterized fragments. Interestingly, for the mesh shader variants, vertex shader launches are also recorded, probably referring to the mesh shader stage. The number of launches for the mesh shader variant that outputs triangles is the same as for the point primitive variant because we launch based on input particles. The difference between the variants is that a larger output allocation is required in the triangle case. Note that each vertex shader invocation fetches a sphere position (and possible color) from storage, in our case, based on SSBOs. Therefore, reducing the number of vertex shader invocations decreases the amount of possibly redundant data fetches. The consequences can be seen in the number of issued load/store instructions (see Figure 14). Among the polygonal pipelines, following intuition, the triangles variant has the highest count and the quad variant the lowest. The geometry shader and mesh shader pipelines behave counterintuitively. We assume this is because they emit vertices and their attributes, and we cannot distinguish between reading and writing data with this metric. The point primitive pipelines issue the lowest number of load/store instructions. This can explain the overall performance gap between point geometry and polygonal billboards. It emphasizes the value of cooperative computational models in stages, such as the mesh shader. However, it underscores that the attribute output

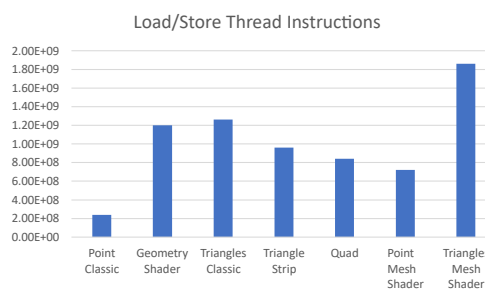


Figure 14: The number of load/store instructions issued in the running threads. As expected, the point primitive has the lowest count, and the polygonal billboards have a ranking following their vertex count. The geometry and mesh shader are exceptions since they emit vertices together with vertex attributes and this metric does not distinguish between read and write instructions. Results recorded on system 1.

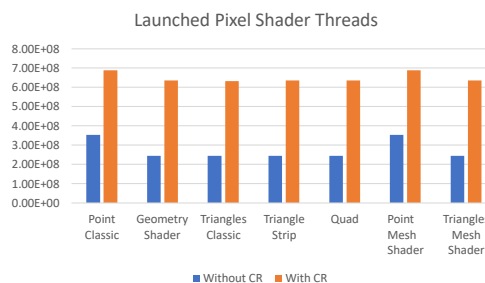


Figure 15: The number of launched fragment shader threads. On the left without active conservative rasterization extension and on the right with active extension. As expected, with the extension, the number of threads is higher and the difference between the pipelines is smaller. Results recorded on system 1.

of the geometry shader and the mesh shader should be as small as possible.

The results discussed in Section 5.2 show that activating the conservative rasterization extension increases rendering time, especially for polygonal billboards and at larger rendering distances to the dataset, hinting at potentially many small triangles that are culled without conservative rasterization. The argument in favor of this extension is that even these small triangles, billboards for the rendered spheres, are important to retain to show the density of a dataset (at least in a binary sense of there is matter in the area). The number of fragments that are finally written in the render target supports this assumption, while the number of fragments remains the same for point primitives with or without the extension, the number of fragments written for polygonal billboards increases, being equal or close to the point primitive counterpart. However, the increase is several magnitudes smaller than the overall fragment count, such that the benefit is doubtful compared to the performance overhead incurred by this extension, for instance, due to the increase in launched fragment shader threads shown in Figure 15.

6. Conclusion

In this evaluation, we compare different rasterization-based rendering pipelines for raycasting spheres. The basic idea of utilizing billboards for this type of rendering is presented by Gumhold [Gum03]. So far, point primitives are considered to provide the best rendering performance results. However, they have the disadvantage that their maximal size is limited in typical rendering APIs, and the limits differ between vendors. This can lead to visual artifacts while closely exploring particle datasets. Our evaluation is focused on the rendering performance at the draw call level. It shows that the performance of triangulated billboards that do not have size limitations can, for specific vendors, be within a sensible margin to the point-based pipelines. However, the point primitive is still the preferable choice. In legacy OpenGL, the quad primitive is the best alternative to the point primitive. However, this primitive type is deprecated, and if available, the mesh shader provides the best performance in the group of polygonal pipelines. The mesh shader is no valid alternative for point primitives.

We have shown that different data layouts have no significant impact on rendering performance. Together with a custom shader preprocessor, this allows the adaptation of the shader programs to many potential input data layouts defined by third-party software. This reinforces the rasterization-based pipelines as “brute-force” renderers for a quick glance at arbitrary data or as a good choice for scenarios where a constant stream of input data or a direct mapping of data into the renderer is expected, such as in situ visualization.

However, in scenarios where pre-processing of input data is allowable, hardware-accelerated ray tracing provides the best rendering performance and quality.

Acknowledgments

This work was partially funded by the German Bundesministerium für Bildung und Forschung (BMBF) as part of project “WindHPC” (In Windkraftanlagen integrierte Second-Life-Rechencluster, 16ME0608K) and by Intel Corporation as part of the Intel oneAPI Center of Excellence program (ID 1510).

Appendix: Implementation Details

Listing 1: Camera-aligned billboard creation.

```
vec3 oc_pos = objPos - camPos;
float sqrRad = rad * rad;

float dd = dot(oc_pos, oc_pos);
float s = sqrRad / dd;
float vi = rad / sqrt(1.0f - s);

vec3 vr = normalize(cross(oc_pos, camUp)) * vi;
vec3 vu = normalize(cross(oc_pos, vr)) * vi;

vec4 v[4];
v[0] = MVP * vec4(objPos - vr - vu, 1.0f);
v[1] = MVP * vec4(objPos + vr - vu, 1.0f);
v[2] = MVP * vec4(objPos + vr + vu, 1.0f);
v[3] = MVP * vec4(objPos - vr + vu, 1.0f);
```

Listing 2: Mesh Shader Main.

```
layout(local_size_x = 31) in;
layout(max_vertices = 31, max_primitives = 31,
       points) out;

out Point {
    flat vec4 pointColor;
    flat vec3 oc_pos;
    flat float rad;
} pp[];

void main() {
    uint g_idx = gl_GlobalInvocationID.x;
    if (g_idx < num_points) {
        uint l_idx = gl_LocalInvocationID.x;

        access_data(g_idx, objPos,
                    pp[l_idx].pointColor, pp[l_idx].rad);
        pp[l_idx].oc_pos = objPos - camPos;
        billboard(objPos, pp[l_idx].rad,
                 pp[l_idx].oc_pos, projPos, 1);

        gl_MeshVerticesNV[l_idx].gl_Position =
            projPos;
        gl_MeshVerticesNV[l_idx].gl_PointSize = 1;

        gl_PrimitiveIndicesNV[l_idx] = l_idx;
    }
    gl_PrimitiveCountNV = min(num_points -
                              gl_WorkGroupID.x * gl_WorkGroupSize.x,
                              gl_WorkGroupSize.x);
}
```

References

- [BTS*16] BERGER M., TAGLIASACCHI A., SEVERSKY L. M., ALLIEZ P., GUENNEBAUD G., LEVINE J. A., SHARF A., SILVA C. T.: A Survey of Surface Reconstruction from Point Clouds. *Computer Graphics Forum* 36, 1 (2016), 301–329. doi:10.1111/cgf.12802. 2
- [ETR19] EISFELD E., TREBIN H.-R., ROTH J.: A wide-range modeling approach for the thermal conductivity and dielectric function of solid and liquid aluminum. *The European Physical Journal Special Topics* 227, 14 (2019), 1575–1590. doi:10.1140/epjst/e2019-800165-5. 5
- [EV15] ECKELSBACH S., VRABEC J.: Fluid phase interface properties of acetone, oxygen, nitrogen and their binary mixtures by molecular simulation. *Physical Chemistry Chemical Physics* 17, 40 (2015), 27195–27203. doi:10.1039/c5cp03415a. 5
- [FGKR16] FALK M., GROTTTEL S., KRONE M., REINA G.: *Interactive GPU-based Visualization of Large Dynamic Particle Data*, vol. 4. Morgan & Claypool Publishers LLC, 2016. doi:10.2200/s00731ed1v01y201608vis008. 1, 2, 3, 4
- [FKE13] FALK M., KRONE M., ERTL T.: Atomistic Visualization of Mesoscopic Whole-Cell Simulations Using Ray-Casted Instancing. *Computer Graphics Forum* 32, 8 (2013), 195–206. doi:10.1111/cgf.12197. 2
- [FSS13] FARIA N., SILVA R., SOBRAL J. L.: Impact of Data Structure Layout on Performance. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2013), IEEE, p. 116–120. doi:10.1109/PDP.2013.24. 3
- [FSW09] FRAEDRICH R., SCHNEIDER J., WESTERMANN R.: Exploring the Millennium Run - Scalable Rendering of Large-Scale Cosmological

- Datasets. *IEEE Transactions on Visualization and Computer Graphics* 15, 6 (2009), 1251–1258. doi:10.1109/tvcg.2009.142. 2
- [GBB*19] GRALKA P., BECHER M., BRAUN M., FRIESS F., MÜLLER C., RAU T., SCHATZ K., SCHULZ C., KRONE M., REINA G., ERTL T.: MegaMol – a comprehensive prototyping framework for visualizations. *The European Physical Journal Special Topics* 227, 14 (2019), 1817–1829. doi:10.1140/epjst/e2019-800167-5. 3
- [GKM*15] GROTTTEL S., KRONE M., MÜLLER C., REINA G., ERTL T.: MegaMol—A Prototyping Framework for Particle-Based Visualization. *IEEE Transactions on Visualization and Computer Graphics* 21, 2 (2015), 201–214. doi:10.1109/tvcg.2014.2350479. 2
- [GKSE12] GROTTTEL S., KRONE M., SCHARNOWSKI K., ERTL T.: Object-Space Ambient Occlusion for Molecular Dynamics. In *2012 IEEE Pacific Visualization Symposium* (2012), IEEE. doi:10.1109/pacificvis.2012.6183593. 2
- [GP07] GROSS M., PFISTER H. (Eds.): *Point-Based Graphics*. Elsevier, 2007. doi:10.1016/b978-0-12-370604-1.x5000-7. 2
- [GRDE10] GROTTTEL S., REINA G., DACHSBACHER C., ERTL T.: Coherent Culling and Shading for Large Molecular Dynamics Visualization. *Computer Graphics Forum* 29, 3 (2010), 953–962. doi:10.1111/j.1467-8659.2009.01698.x. 2, 7
- [GRE09] GROTTTEL S., REINA G., ERTL T.: Optimized Data Transfer for Time-dependent, GPU-based Glyphs. In *2009 IEEE Pacific Visualization Symposium* (2009), IEEE. doi:10.1109/pacificvis.2009.4906839. 2
- [GRE23] GRALKA P., REINA G., ERTL T.: Supplemental Data on Measured Rendering Times, 2023. doi:10.18419/darus-3458. 6
- [Gum03] GUMHOLD S.: Splatting Illuminated Ellipsoids with Depth Correction. In *Proceedings of Workshop on Vision, Modelling, and Visualization 2003* (2003), pp. 245–252. 1, 2, 10
- [GWG*20] GRALKA P., WALD I., GERINGER S., REINA G., ERTL T.: Spatial Partitioning Strategies for Memory-Efficient Ray Tracing of Particles. In *2020 IEEE 10th Symposium on Large Data Analysis and Visualization* (2020), IEEE. doi:10.1109/ldav51489.2020.00012. 1, 2, 3, 7
- [HE03] HOPF M., ERTL T.: Hierarchical Splatting of Scattered Data. In *IEEE Visualization, 2003* (2003), IEEE. doi:10.1109/visual.2003.1250404. 2
- [HHVM20] HITZ T., HEINEN M., VRABEC J., MUNZ C.-D.: Comparison of macro- and microscopic solutions of the riemann problem i. supercritical shock tube and expansion into vacuum. *Journal of Computational Physics* 402 (2020), 109077. doi:10.1016/j.jcp.2019.109077. 5
- [IRR*22] IBRAHIM M., RAUTEK P., REINA G., AGUS M., HADWIGER M.: Probabilistic Occlusion Culling using Confidence Maps for High-Quality Rendering of Large Particle Data. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2022), 573–582. doi:10.1109/tvcg.2021.3114788. 2
- [Khr21] KHROSOS GROUP: glslang, 2021. <https://github.com/KhronosGroup/glslang> (accessed: 2021-12-10). 3
- [LBH12] LINDOW N., BAUM D., HEGE H.-C.: Interactive Rendering of Materials and Biological Structures on Atomic and Nanoscopic Scale. *Computer Graphics Forum* 31, 3pt4 (2012), 1325–1334. doi:10.1111/j.1467-8659.2012.03128.x. 2
- [MBE19] MÜLLER C., BRAUN M., ERTL T.: Optimised Molecular Graphics on the HoloLens. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)* (2019), IEEE. doi:10.1109/vr.2019.8798111. 2
- [Meg23] MEGAMOL DEV TEAM: MegaMol Repository, 2023. <https://github.com/UniStuttgart-VISUS/megamol> (accessed 2023-05-05). 3
- [NBB*14] NIETHAMMER C., BECKER S., BERNREUTHER M., BUCHHOLZ M., ECKHARDT W., HEINECKE A., WERTH S., BUNGARTZ H.-J., GLASS C. W., HASSE H., VRABEC J., HORSCH M.: ls1 mardyn: The Massively Parallel Molecular Dynamics Code for Large Systems. *Journal of Chemical Theory and Computation* 10, 10 (2014), 4455–4464. doi:10.1021/ct500169q. 8
- [NVI22] NVIDIA: Introduction to Turing Mesh Shaders, 2022. <https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/> (accessed: 2022-03-07). 5
- [NVI23] NVIDIA: NVIDIA Nsight Perf SDK, 2023. <https://developer.nvidia.com/nsight-perf-sdk> (accessed 2023-03-13). 8
- [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 Transactions on Graphics* 29, 4 (2010), 1–13. doi:10.1145/1778765.1778803. 3, 7
- [PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum* 26, 3 (2007), 415–424. doi:10.1111/j.1467-8659.2007.01064.x. 2
- [PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LVBH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *High Performance Graphics* (2010), The Eurographics Association, pp. 87–95. doi:10.2312/EGGH/HPG10/087-095. 2
- [RHI*15] RIZZI S., HERELD M., INSLEY J., PAKKA M. E., URAM T., VISHWANATH V.: Large-Scale Parallel Visualization of Particle-Based Simulations using Point Sprites and Level-Of-Detail. In *Eurographics Symposium on Parallel Graphics and Visualization* (2015), The Eurographics Association. doi:10.2312/pgv.20151149. 2
- [RL00] RUSINKIEWICZ S., LEVOY M.: QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00* (2000), ACM Press. doi:10.1145/344779.344940. 2
- [SGG15] STAIB J., GROTTTEL S., GUMHOLD S.: Visualization of Particle-based Data with Transparency and Ambient Occlusion. *Computer Graphics Forum* 34, 3 (2015), 151–160. doi:10.1111/cgf.12627. 2
- [Wie21] WIEDEMANN M.: *Modeling Concurrent Data Rendering and Uploading for Graphics Hardware*. PhD thesis, 2021. doi:10.5282/EDOC.27979. 1, 3
- [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 (2017), 931–940. doi:10.1109/tvcg.2016.2599041. 3
- [WK19] WIEDEMANN M., KRANZLMÜLLER D.: Statistical Analysis of Parallel Data Uploading using OpenGL. *Eurographics Symposium on Parallel Graphics and Visualization* (2019). doi:10.2312/pgv.20191114. 3
- [WKJ*16] WALD I., KNOLL A., JOHNSON G. P., USHER W., PASCUCCI V., PAKKA M. E.: CPU Ray Tracing Large Particle Data with Balanced P-k-d Trees. In *2015 IEEE Scientific Visualization Conference, SciVis 2015 - Proceedings* (2016), IEEE, pp. 57–64. doi:10.1109/SciVis.2015.7429492. 1, 2, 7
- [ZAMW20] ZELLMANN S., AUMÜLLER M., MARSHAK N., WALD I.: High-Quality Rendering of Glyphs Using Hardware-Accelerated Ray Tracing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2020), The Eurographics Association. doi:10.2312/pgv.20201076. 1, 2