Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

# Spatial CPU-GPU Data Structures for Interactive Rendering of large Particle Data

Sergej Geringer

| | |
|---|---|
| **Course of Study:** | Informatik |
| **Examiner:** | Prof. Dr. Thomas Ertl |
| **Supervisor:** | Dr. Guido Reina<br>Patrick Gralka, M.Sc. |
| **Commenced:** | January 20, 2017 |
| **Completed:** | July 20, 2017 |
| **CR-Classification:** | I.3, I.3.6, I.3.7 |

# Kurzfassung

In dieser Arbeit wird die interaktive Visualisierung beliebig großer Partikeldaten untersucht, wobei die Partikeldaten im Arbeitsspeicher hinterlegt sind, aber nicht zwangsläufig in den Grafikspeicher passen. Mit üblichen Rendering Methoden büßen Visualisierungen drastisch an Interaktivität ein, wenn mehrere zehn- bis hunderte Millionen Objekte dargestellt werden. Gleichzeitig ist die Größe möglicher zu visualisierender Datensätze begrenzt durch den Videospeicher von Grafikkarten, auf dem zu visualisierende Daten vorliegen müssen. Um diese Einschränkungen zu umgehen, wird in dieser Arbeit ein progressiver Rendering Ansatz verfolgt, der sukzessive alle Partikeldaten zur Grafikkarte hochlädt und rendert, ohne die Partikeldaten zu reduzieren oder anderweitig zu verändern. Die Partikeldaten werden entsprechend einer vorgenommenen Sichtbarkeitssortierung gerendert, die aus gegenseitigen Verdeckungen verschiedener Teile des Partikeldatensatzes berechnet wird. Dies führt dazu, dass Teile der Szene nach ihrer Wichtigkeit für das aktuelle Bild sortiert und dargestellt werden.

Es werden verschiedene Möglichkeiten analysiert und verglichen, Partikel als opake Kugeln in OpenGL zu rendern. Dies formt die Grundlage für die Partikel-Rendering Software, die in dieser Arbeit entwickelt wurde. Die Architektur der Rendering-Software benutzt mehrere Threads, sodass durch eine Daten-Vorverarbeitung auf einem CPU-Thread und durch Rendering-Algorithmen auf einem GPU-Thread sichergestellt ist, dass der Benutzer mit der Software jederzeit interagieren kann. Insbesondere ist sichergestellt, dass der Benutzer die Partikeldaten interaktiv untersuchen kann, indem die Latenz zwischen Benutzereingaben und dem Anzeigen der daraus resultierenden Veränderungen minimal gehalten wird. Dies wird erreicht indem der Verarbeitung von Benutzereingaben an allen Stellen des Rendering-Prozesses höhere Priorität eingeräumt wird als der Vollständigkeit des gerenderten Bildes. Gleichzeitig wird dem Benutzer eine sofortige Rückmeldung über getätigte Benutzereingaben gegeben, indem alle sichtbaren Partikel in das nächste gerenderte Bild neu projeziert werden. Diese Neu-Projektion wird durch einen GPU-seitigen Partikel-Cache aller aktuell sichtbaren Partikel realisiert, der während des sukzessiven Partikelstreamings und -renderns aufgebaut wird. Sobald der Benutzer eine Eingabe tätigt, wird der auf der GPU liegende Partikel-Cache unter der aktuellsten benutzerdefinierten Kameraposition neu gerendert.

Die Kombination dieser entwickelten Methoden erlaubt ein interaktives Betrachten von Partikeldaten mit bis zu $1,5$ Milliarden Partikeln auf einem handelsüblichen Computer.

# Abstract

In this work, I investigate the interactive visualization of arbitrarily large particle data sets which fit into system memory, but not into GPU memory. With conventional rendering techniques, interactivity of visualizations is drastically reduced when rendering tens- or hundreds of millions of objects. At the same time, graphics hardware memory capabilities limit the size of data sets which can be placed in GPU memory for rendering. To circumvent these obstacles, a progressive rendering approach is employed, which gradually streams and renders all particle data to the GPU without reducing or altering the particle data itself. The particle data is rendered according to a visibility sorting derived from occlusion relations between different parts of the data set, leading to a rendering order of scene contents guided by importance for the rendered image.

I analyze and compare possible implementation choices for rendering particles as opaque spheres in OpenGL, which forms the basis of the particle rendering application developed within this work. The application utilizes a multi-threaded architecture, where data preprocessing on a CPU-thread and a rendering algorithm on a GPU-thread ensure that the user can interact with the application at any time. In particular it is guaranteed that the user can explore the particle data interactively, by ensuring minimal latency from user input to seeing the effects of that input. This is achieved by favoring user inputs over completeness of the rendered image at all stages during rendering. At the same time the user is provided with an immediate feedback about interactions by re-projecting all currently visible particles to the next rendered image. The re-projection is realized with an on-GPU particle-cache of visible particles that is built during particle data streaming and rendering, and drawn upon user interaction using the most recent camera configuration according to user inputs.

The combination of the developed techniques allows interactive exploration of particle data sets with up to $1.5$ billion particles on a commodity computer.

# Contents


1 Introduction 17
- 1.1 Motivation . . . 17
- 1.2 Related Work . . . 18
- 1.3 Research Question and Structure of Thesis . . . 21

2 Rendering Particles with OpenGL 25
- 2.1 Ray Casting Sphere Glyphs . . . 27
  - 2.1.1 Basic Algorithm . . . 27
  - 2.1.2 OpenGL Implementation . . . 30
  - 2.1.3 Performance Measurements and Implications . . . 36
- 2.2 Ray Casting pkd-Trees on the GPU . . . 47
  - 2.2.1 Algorithm on the GPU . . . 48
  - 2.2.2 Performance Measurements and Implications . . . 53

3 Organizing Large Data Sets 57
- 3.1 Using a Uniform Grid . . . 59
  - 3.1.1 Grid Construction . . . 59
  - 3.1.2 Frustum Culling . . . 63
- 3.2 Visibility and the Occlusion Graph . . . 67
  - 3.2.1 Computing Occlusion in the Grid . . . 70
  - 3.2.2 Deriving a Sorting of the Scene . . . 73

4 Interactive Rendering of Large Particle Data 79
- 4.1 OpenGL Renderer Design . . . 81
  - 4.1.1 Data Loading . . . 81
  - 4.1.2 Multi-Threaded Rendering . . . 81
  - 4.1.3 Particle Rendering . . . 83
- 4.2 Progressive Rendering and Data Streaming . . . 87
- 4.3 Utilizing a GPU Particle Cache . . . 90

5 Results 95
- 5.1 Evaluating Interactivity . . . 97
- 5.2 Evaluating Progressive Rendering and Visibility Sorting . . . 101

5.3 Evaluating Particle Cache Impact . . . . . 108
6 Conclusion and Future Work 113
Bibliography 117

# List of Figures


2.1 Ray casting spheres in OpenGL. . . . . 28
2.2 Computing the extent of proxy geometry. . . . . 29
2.3 Test data sets for particle rendering methods. . . . . 36
2.4 Memory layout of kd-tree and resulting spatial relations of particles. . . 49
2.5 Ray traversal in the pkd-tree. . . . . 51
2.6 Pkd-tree test data sets. . . . . 53

3.1 Partitioning of particle data into grid cells. . . . . 60
3.2 View-frustum culling of spheres in clip space. . . . . 65
3.3 Illustrating the importance of visibility-based object sorting. . . . . 68
3.4 Occlusion between bricks in a grid. . . . . 71
3.5 Approximating occlusion between bricks by ray-grid traversal. . . . . 72
3.6 Occlusion graph resulting from brick occlusions in a grid. . . . . 73
3.7 Occlusion graph and resulting node levels according to distance from the camera. . . . . 75
3.8 Density-based brick visibility sorting using the occlusion graph. . . . . 77

4.1 Comparison of points-only and spheres-only rendering of particles. . . . 84
4.2 Effect of the on-GPU particle cache during user interaction. . . . . 90
4.3 G-buffer reduction and particle cache filling scheme. . . . . 92

5.1 Renderings of data sets used for evaluation. . . . . 95
5.2 Streaming evaluation of visibility sortings for *Halle10* scene. . . . . 105
5.3 Streaming evaluation of visibility sortings for *LaserBig* scene. . . . . 106
5.4 Streaming evaluation of visibility sortings for *LaserCharge* scene. . . . . 107
5.5 Impact of on-GPU particle-cache on the Halle10 scene in Overview view. 109
5.6 Impact of on-GPU particle-cache on the Halle10 scene in Detail view. . . 110
5.7 Impact of on-GPU particle-cache on the LaserBig scene in Detail view. . . 111

# List of Tables

2.1 Times of *APU* system for different sphere rendering methods. . . . . 40
2.2 Times of *APU* system for data streaming. . . . . 40
2.3 Times of *TITAN* system for different sphere rendering methods. . . . . 41
2.4 Times of *TITAN* system for data streaming. . . . . 42
2.5 Times of the *TITAN* system for rendering particle data sets of different sizes. 43
2.6 Times of *RadeonR9* system for different sphere rendering methods. . . . 44
2.7 Times of *RadeonR9* system for data streaming. . . . . 44
2.8 Times of the *RadeonR9* system for rendering particle data sets of different sizes. . . . . 45
2.9 Pkd-Tree traversal times on the TITAN and Radeon R9. . . . . 56
2.10 Traversing different pkd-tree sizes on the Radeon R9. . . . . 56

5.1 Frame update times for different data sets, grid sizes and streaming configurations. . . . . 100

# List of Listings

4.1 Single-threaded render loop. . . . . 82
4.2 Multi-threaded render loop. . . . . 82
4.3 G-buffer structure for deferred rendering. . . . . 85
4.4 G-buffer structure holding particle data. . . . . 91

# List of Algorithms

3.1 Level assignment to nodes in the occlusion graph. . . . . 74

# 1 Introduction

## 1.1 Motivation

In this work, the interactive visualization of large particle- and point-data sets is studied. Particle data is produced by measurements or simulations in a variety of fields like molecular dynamics, physical cosmology or material sciences. Particles represent distinct objects which have a position in space. Depending on the application field, particles may have further attributes like an orientation, color, mass, or velocity, and in simulations those attributes may influence behavior and attributes of nearby particles, leading to complex systems of particles interacting with each other. Similar to particles, points and point cloud data can stem from acquisition devices like laser scanners or be constructed from images using computer vision techniques. Point clouds represent surfaces of (measured) objects, without explicitly storing the connectivity of the surface points, but with further attributes like surface-normals, color or distance to the measuring device. Due to factors like growing computational power and efficiency in simulations, increased sensor accuracy and cheaper storage, particle data as well as point clouds with ever-increasing size are produced, which presents new challenges to the evaluation and analysis of such data sets.

Interactive visualization tools assist researchers and domain experts to draw insights from complex data sets by allowing them to explore the data in an immediate and comprehensible fashion. This way, properties of the data set can be evaluated quickly and further data analysis can be guided by those insights. While the actual representation of the data strongly depends on the application domain (according to the data analysis, filtering and mapping steps of the visualization pipeline), rendering algorithms need to ensure a fast rendering of the prepared data. Most rendering algorithms nowadays utilize the graphics processing unit (GPU) to implement highly optimized rendering techniques to allow fast, interactive visualization. *Interactivity* allows the user to interact with the visualization tool and get an immediate response of the application to user inputs, e.g. looking at a different part of the data set by moving the computer mouse and immediately being presented with the rendered image resulting from that action. However, with increasing size of particle data, rendering algorithms not only have to manage correct rendering, but they have to ensure timely rendering of data sets which

are too large to fit into GPU memory resources, and sometimes too large to fit into overall system memory.

While data set sizes increase, the capabilities of modern GPUs can not keep up with the task of rendering that data immediately. Large particle data sets can easily grow too large to entirely fit into GPU memory, which is where data needs to reside before being rendered by the GPU. Furthermore, data upload to the GPU is not arbitrarily fast, but it is limited by hardware and software circumstances like the PCIe bus and the GPU driver managing data transfers. Even if those memory limitations are ignored, rendering still needs to occur fast enough to ensure interactivity, and although GPUs offer a tremendous processing power, rendering tens- or hundreds of millions of particles on the GPU can easily take much longer than is acceptable for an interactive visualization.

So due to increasingly larger data sets and insufficient capabilities of modern GPUs, the interactive rendering of large particle data is a non-trivial task. To somehow visualize particle data with several hundreds of millions of individual particles, the data not only needs to be managed and rendered efficiently on the GPU, but it must be organized and provided to the GPU such that important data is available when needed. At the same time, the visualization application should provide the user with a correctly rendered image while still being interactive, meaning that the user should be able to make inputs to the application and get a correct feedback immediately.

## 1.2 Related Work

*Point cloud data* stemming from scanning devices can be processed in a number of ways for visualization. Focusing on fast rendering and compact data representation, one of the first point data visualization techniques is QSplat [RL00], which interactively visualizes scanned point data from the Digital Michelangelo Project [LPC+00] by managing the data in a multi-resolution bounding sphere hierarchy and storing a level-of-detail representation of data at hierarchy levels. High-quality surface reconstruction for point data is achieved by Surface Splatting [ZPBG01] and Surfels [PZBG00], which render data points as local surface patches, and smooth or blend them over a local area of influence to achieve the representation of a connected high-quality surface rendering. Such techniques can be approximated on graphics hardware [BHZK05; BK03], thus enabling GPU-accelerated high-quality point based rendering. To handle rendering of large point data sets interactively, multi-resolution hierarchies combined with level-of-detail approximations and compression of the data set are used [SPL04]. Sequential Point Trees [DVS03] utilize an octree data structure where a tree node holds disk-geometry and subtrees in child-nodes. The disk locally approximates the surface represented by the particle data in the child-nodes. During rendering, the hierarchy is traversed and

depending on an error metric, a node's disk gets drawn or the child-nodes get recursively traversed until a small enough image-error is established when drawing the subtrees. The memory layout of the tree is organized in a way such that it can be efficiently used for rendering on the GPU after the CPU determined to-be-rendered parts of the data set during a tree traversal. Instead of approximating point data locally, Layered Point Clouds [GM04] build a multi-resolution hierarchy by using a subset of all data points as data representation for a tree node at a hierarchy level, then split the remaining data into two spatially disjoint parts which are used as children-nodes of that level. In that manner, a hierarchy is built in which each level holds refinements of preceding hierarchy levels, but instead of holding approximations of data points, subsets of the data itself are stored at each hierarchy level.

Because building of good-quality hierarchies can take time and thus delay visual evaluation of point data sets, algorithms which can render point data with less preprocessing are proposed by [BDS05; WS06]. [BDS05] build simplified geometry meshes with high quality normal maps derived from the point data for fast rendering on GPU hardware, while [WS06] combine Sequential Point Trees with nested octrees and streaming of out-of-core data which allows for interactive navigation through unprocessed point data sets with up to $262$ million points. Out-of-core rendering algorithms combined with level-of-detail hierarchies and data compression/quantization become mandatory for point cloud rendering with ever increasing data set sizes of unstructured point data [EBN13; GZPG10; PGA11; RD10], realizing interactive visualization of $1-5$ billion data points stemming from scans of landscapes, cities or cultural heritage sites. [PGA11] extract surface representations on the GPU for shading of unstructured point clouds and [TUH+14] employ stochastic point-based rendering to render point data transparently on the GPU.

*Cosmological and astronomical particle data* stems from large-scale universe evolution simulations with billions of data points. To visualize volumetric large-scale structures, [HE03] employ a splatting of hierarchical particle clusters which are determined by a principal component analysis clustering algorithm. [FSW09] use a multi-resolution octree structure which holds a level-of-detail hierarchy to allow interactive rendering of the Millennium Run Simulation with more than $10$ billion particles, where they render particles at a certain level-of-detail by splatting sprites on the GPU. To preserve visual details, hybrid approaches visualize such data sets using direct volume rendering combined with direct rendering of particles to achieve interactive exploration of data sets with up to one trillion particles [Lpa14; SMK+16], utilizing multiple GPUs to distribute workloads. Due to the huge size of data sets, network streaming of data that is out-of-core and aggressive compression are employed. Visualization systems for cosmological simulations combine such rendering techniques with tools that allow interactive analysis of time dependent formation of dark matter halo structures on commodity hardware [PGX+16; SBD+15]. [RHI+15] report the implementation of a

framework that allows to visualize up to $32$ billion particles on a cluster of 128 GPUs by splatting point sprites.

*Dynamic particle data from fluid simulations* can be interactively visualized by sampling particles into volumetric grid- or hierarchical voxel-representations and evaluating those structures using volume ray-casting [HOK16; RCSW14; ZD15].

*Particle data from molecular dynamics* is usually visualized on the GPU by rendering particles according to some natural representation like sphere glyphs [GSGP06], some surface representation derived from structural properties of molecules [KBE09], or other representations suitable for analysis of data [KKL+15; RE05]. Reducing data size of large atomic or molecular data sets to maintain interactivity during rendering is possible by data compression when important structural properties of the data set are guaranteed to stay intact [GGRE13], or by level-of-detail methods when the structure of molecules is known and can be preserved by some error metric during simplification [GNL+15; PJR+14]. Knowledge about reappearing identical molecular structures can be exploited to accelerate rendering and reduce memory pressure on system memory and graphics hardware by using instancing [FKE13; LBH12]. [GRDE10] employ occlusion culling using GPU-occlusion-queries to avoid drawing expensive geometry and achieve interactive visualization for large molecular data with up to $100$ million particles where no structural information is given beforehand. Using optimized spatial data structures and algorithms, ray tracing massive molecular data sets on multi-core CPU architectures is shown to be feasible at interactive frame rates by [GIK+07; KWN+13; WKJ+15]. With such CPU-ray tracing frameworks, advanced illumination techniques like inter-reflections, transparency and ambient-occlusion are easy to realize [WJA+17], while for GPU-based implementations such features need techniques adopted to the computational model of graphics hardware [GKSE12; GRDE10; SGG15; SVGR16]. However, interactive molecular visualization on commodity hardware is easier to realize using the computational power of GPUs [GKM+15; MKB+15], which mostly will be cheaper than CPU architectures allowing interactive rendering with ray tracing. With growing data set sizes however, it becomes more difficult to adopt rendering algorithms on GPUs to interactively handle large amounts of particle data.

## 1.3 Research Question and Structure of Thesis

The goal of this work is to evaluate and develop spatial data- and acceleration-structures which enable rendering of large particle data. Ideally, such data structures should support efficient evaluation on the CPU and GPU at the same time. While the CPU-processing of the data set should be able to determine which parts of the particle data should be rendered first, the final particle drawing on the GPU should benefit from the design of the data structures to allow fast and thus interactive rendering of the particle data. Because the particle data is assumed to not fit into GPU memory, it is important to develop data structures which benefit both the CPU-side preprocessing as well as the GPU-side rendering, to enable rendering of massively large data sets. The nature of particle data sets may involve dense and non-dense structures on different scales. This makes adoption of occlusion culling techniques difficult, as dense or scattered clusters of particles can not be trivially tested to occlude other parts of the data set. Thus, management and rendering of particle data needs to involve a spatial sorting and prioritization scheme which provides the rendering algorithms with data that is relevant for the current image.

While the rendering of dynamic particle data sets stemming from time dependent simulations is important to support researchers in application domains analyzing that data, the algorithms developed in this work focus on handling one static particle data set at a time. Compared to static data, large dynamic particle data sets worsen the problem of timely data availability for rendering on the GPU. Many of the methods developed in this work can be applied to handle dynamic data sets, but this work focuses its analysis on static data.

Interactive rendering of large particle data can be made feasible when processing load for the GPU is minimized, which is often done by utilizing level-of-detail and hierarchical structures by deriving smaller, simplified representations of the data which fit into GPU memory and ca be rendered fast. But a simplified representation of the particle data strongly depends on the nature and structure of the data set. Algorithms which capture and simplify the structures of particle data sets must therefore rely on assumptions about the data depending on the application domain and properties of particles. So level-of-detail and hierarchical methods for rendering particles are reliable when knowledge about the particle data is available. To render arbitrary particle data however, such approaches are not applicable. Furthermore, computing (hierarchical) abstractions of the data set may take very long time and the resulting renderings do not show the actual data, but only approximations that are bound by some error metric chosen during simplification. For these reasons, level-of-detail or hierarchical simplification algorithms are not considered in this work. Instead, the particle data set will be rendered in its

original form, without any preprocessing or compression that might change the nature of the data.

Rendering particle data involves three main steps of processing. First, the actual rendering of particles involves a rendering algorithm that transforms the abstract particle data (a position in space with further attributes) to some suitable representation on the screen. Secondly, a CPU-side preprocessing of the particle data needs to ensure that important particle data gets extracted and prioritized according to its contribution to the current image. Thirdly, the preprocessed particle data must be transfered to the GPU for rendering in an efficient manner, ensuring interactivity of the application to the user while feeding the low-level rendering algorithms with particle data. These three processing steps are investigated in the main chapters of this thesis, Chapter 2, Chapter 3 and Chapter 4. Chapter 5 evaluates interactivity and rendering performance of the algorithms designed in the previous chapters, followed by a final conclusion of the results in Chapter 6.

Chapter 2: Rendering Particles with OpenGL

In this work, individual particles will be rendered as opaque spheres in space. The low-level rendering algorithm for this is implemented in modern OpenGL $4.5$, a graphics application programming interface supported by all currently available graphics cards and commonly used operating systems. The low-level OpenGL implementation details of the sphere rendering algorithm can drastically influence performance of the application, which is the reason this subject is investigated in Chapter 2. Section 2.1 analyzes the possible implementation choices when implementing sphere ray casting via proxy geometry rasterization in OpenGL. The resulting variations of sphere rendering algorithms with OpenGL are tested with large particle data sets on different GPU hardware to evaluate resulting rendering performance. As an alternative to the rasterization-oriented sphere ray casting, the spatial pkd-tree data structure introduced by [WKJ+15] is ported to an OpenGL implementation in fragment shaders and evaluated in Section 2.2 for its fitness to render large particle data sets. However, evaluations of the pkd-tree data structure on the GPU show that it is not suitable to act as a shared spatial data structure used by preprocessing on the CPU and for rendering on the GPU. Thus, sphere ray casting using rasterized proxy geometry is used as basic OpenGL rendering algorithm for particles.

Chapter 3: Organizing Large Data Sets

Since individually processing millions of particles to decide their importance for the currently rendered image is not feasible on commodity hardware, the data must be

organized in a way that allows to make rendering decisions quickly and efficiently. Chapter 3 investigates organizing particle data into a uniform grid. The grid is used to spatially partition the particle data into smaller groups (bricks), which for example are used to cull large particle groups outside the view frustum (see Section 3.1). Section 3.2 describes how the uniform structure of the grid can be used to derive a visibility sorting of particles in the scene by computing occlusion relations between bricks of particles. The occlusion relations of bricks in the scene are represented in an occlusion graph, from which two kinds of visibility sortings for the data set are derived. The visibility sortings resulting from the occlusion relations between bricks in the scene realize a front-to-back sorting of scene contents that incorporates importance of particle clusters for the currently rendered image. The bricks of particles are then rendered to the screen in the order determined by the visibility sorting.

Chapter 4: Interactive Rendering of Large Particle Data

The important design concepts of the interactive particle visualization application developed in the course of this work are described in Chapter 4. Section 4.1 describes the basic design of the multi-threaded rendering application, which decouples CPU-preprocessing of user inputs from actual rendering algorithms that are running on a dedicated GPU-rendering-thread. Particle data is prepared and visibility-sorted on the CPU-thread upon user interaction and passed to the GPU-thread for rendering. The particles are not rendered all at once, instead a progressive rendering approach gradually builds the rendered image from particle data which is streamed to GPU memory bit by bit. This progressive rendering and data streaming is detailed in Section 4.2. The progressive rendering is designed in a way such that the user can make inputs at any time during rendering, ensuring interactivity by minimizing latency between new user inputs and the point in time when their impact on the visualization can be perceived. To further increase interactivity of the application, an on-GPU particle cache is implemented, which is detailed in Section 4.3. The particle cache holds all currently visible particles and gets drawn as first approximation of scene contents upon user interaction, while missing particle data gets progressively added to the rendered image by the streaming mechanism.

The evaluation of the developed methods in Chapter 5 analyzes interactivity and convergence of the progressive rendering, showing that particle data sets of arbitrary size can be explored interactively and with good image quality. The size of the handled data sets is only limited by system memory.

# 2 Rendering Particles with OpenGL

Depending on the application domain and nature of the data, particles can be rendered in a number of ways. In molecular dynamics, atomic particles can be rendered as sphere glyphs as defined by the van der Waals radii of individual atoms [LR71]. The placement of individual atoms inside a molecule can drastically influence the properties of the molecule, so an accurate rendering of such particles is important for correctness. In contrast, particle data from cosmological simulations represents phenomena on a large scale, where individual particles contribute to the structure of large clusters. To make the rendering of large quantities of particles feasible, particles are sampled into volumetric grids or hierarchical structures, which then are visualized using volume rendering or particle splatting [FSW09; HE03; RTW13]. To retain details which are lost by sampling into coarse volumes, hybrid approaches additionally perform direct particle rendering [Lpa14; SMK+16]. Point cloud rendering techniques visualize data stemming from acquisition devices which scan surfaces. Such point data is rendered as oriented disks which are blended together to appear as a connected surface [BHZK05; GP07]. Since every point contributes to the local appearance of its surface, points can only be left out of the rendering if proper reconstruction of the surface is assured by other means. This leads to level-of-detail approaches where approximate representations of the point data are stored in hierarchical data structures [RL00; WS06].

In the following, rendering of particles as sphere glyphs is going to be considered, without any approximations or level-of-detail reduction of the data. The purpose of this chapter is to analyze the performance characteristics of OpenGL implementation choices when rendering particles as opaque spheres. Therefore, the low-level implementation details of how to render a set of particles in OpenGL are going to be discussed. The reason to investigate this is that for large particle data sets, several millions of particles are going to be rendered each frame. While subtle implementation details may not impact rendering performance for small data sets, rendering speed may drastically depend on such details when increasing data size. Implementation choices concerning GPU memory, geometry processing and fragment operations may lead to different optimization possibilities for both the OpenGL driver and GPU hardware, thus affecting overall rendering performance. Although primarily sphere glyphs are going to be considered, insights into performance characteristics may also apply to rendering of other primitives, since the underlying OpenGL principles will probably not be limited to sphere rendering alone.

The first part of this chapter will look into brute-force rendering of large amounts of sphere glyphs and resulting OpenGL rendering performance. The second part of this chapter will look into an alternative way of rendering spheres, which evaluates a kd-tree data structure entirely on the GPU. The brute-force approach uses the rasterization pipeline to draw lots of geometry which is manipulated to be rendered as spheres into the framebuffer, while the kd-tree approach ports the pkd-tree data structure introduced by [WKJ+15] to the GPU and traverses it in the fragment shader.

## 2.1 Ray Casting Sphere Glyphs

Since the OpenGL rasterization pipeline handles triangles, one way to render a sphere in OpenGL consists of generating a triangle mesh of the sphere and then draw it. However, the triangle mesh is only an approximation of the sphere's actual geometry and depending on certain factors, the mesh will not always look like a perfect sphere. The rendered sphere will not be *pixel perfect*, in the sense that the outline of the sphere will appear to have corners when looked at closely. Quality of the rendering could be improved by further refining the mesh, using more and smaller triangles. But the finer mesh entails more work for OpenGL draw commands and therefore leads to slower rendering, even more so when rendering lots of spheres.

Rendering *pixel perfect* spheres in OpenGL can be done by ray casting the sphere's surface during the fragment shader stage after drawing a very simple proxy geometry. The ray casting decides for each fragment to either write it into the framebuffer if it contributes to the appearance of the sphere, or else to *discard* it from being written to the framebuffer. This method of rendering particles on the GPU by ray casting ellipsoids was first introduced by [Gum03]. Despite the geometry processing and tessellation capabilities of modern graphics hardware, imposing less geometry workload and implicitly rendering surfaces during the fragment shader stage most probably is going to be faster than imposing heavy geometry workload.

### 2.1.1 Basic Algorithm

Let $s = (c, r)$ be a sphere with center $c = (x, y, z)^\intercal \in \mathbb{R}^3$ and radius $r > 0$. The sphere rendering algorithm proceeds in three steps.

1. Draw a simple proxy geometry mesh that covers the area in screen space where the sphere is to be rendered.
2. For each fragment generated by the draw call: from the fragment's window coordinates $f_{window}$ find its position in world space $f_{world} = f \in \mathbb{R}^3$ (or in some other suitable space, e.g. view space). Given a camera position $v \in \mathbb{R}^3$ in world space, construct a ray $r = \left(v, \overrightarrow{vf}\right)$ from the camera to the fragment [1].
3. For each fragment generated: test the ray $r$ and sphere $s$ for intersection. If the ray does not intersect the sphere, do not write the current fragment to the framebuffer (*discard*). If the ray intersects the sphere, write the fragment to the framebuffer

[1] For vectors $a, b \in \mathbb{R}^3$ define $\overrightarrow{ab} = b - a$.

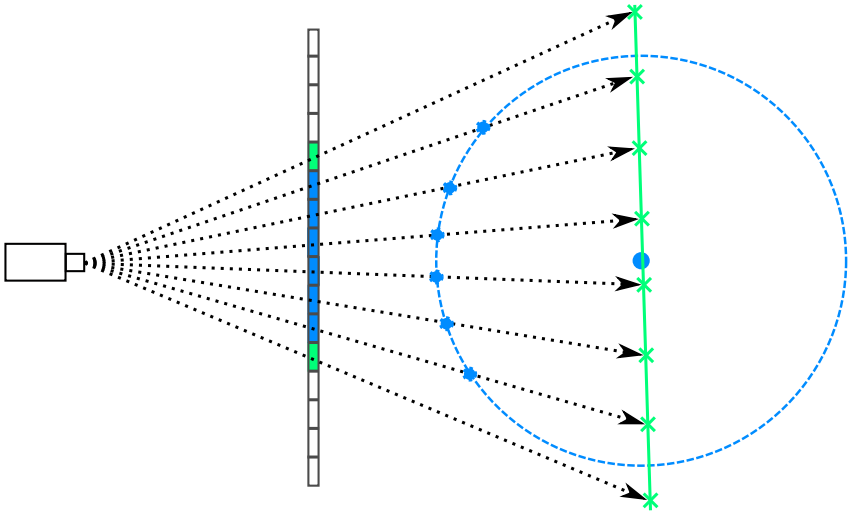


**Figure 2.1:** Ray casting a sphere in OpenGL. From left to right: scene camera, image plane with fragments, proxy geometry and targeted sphere geometry. A proxy geometry (green line) is drawn to spawn fragment shader invocations. The view rays induced by the generated fragments are tested for intersection with the sphere. Fragments for which the ray does not intersect the sphere will be discarded (green fragments). Rays intersecting the sphere provide a new depth value (blue dots on the sphere) for the fragment, which is written to the depth buffer instead of the original depth value of the proxy geometry.

with a new depth buffer value according to the distance of the intersection point to the camera.

The proxy geometry can be a mesh with very few triangles. It suffices to draw a rectangle consisting of two triangles, as long as the rectangle fully covers the screen space area where the sphere is to be rendered. The fragments resulting from drawing the proxy geometry are used to initialize a primary view ray, which then is used for analytically ray casting sphere geometry. This way a pixel perfect sphere can be rendered with minimal geometry processing cost. Figure 2.1 illustrates the algorithm. It is crucial to write the correct depth value of the ray-sphere intersection point into the depth buffer, or else the resulting image will be incorrect when multiple objects occlude each other.

To minimize the workload on the GPU, it is desirable to generate as few discarded fragments as possible. This implies choosing the proxy geometry as small as possible while maintaining a full coverage of the sphere's screen-footprint.

To derive the correct extent for the proxy geometry, consider the setup depicted in Figure 2.2. Given the sphere $s = (c, r) \in \mathbb{R}^3 \times \mathbb{R}_{>0}$ we want to find the point $g$ on the outline contour of the sphere as seen by the viewer. Assume the viewer position to be at

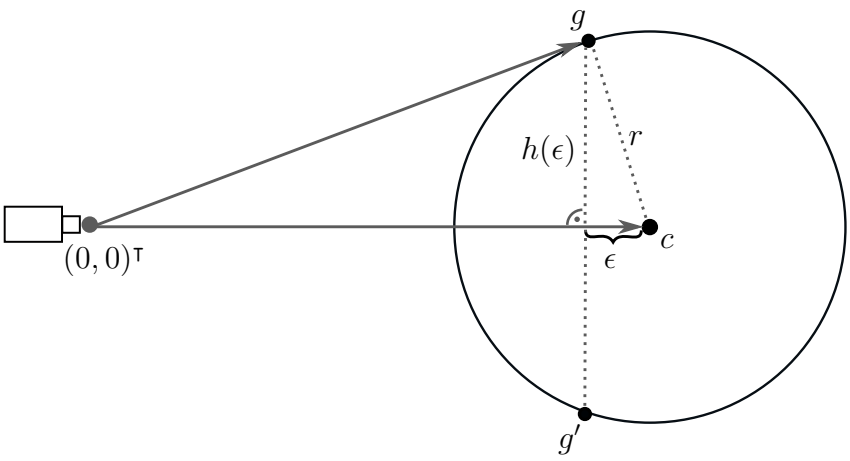



**Figure 2.2:** Computing the extent of proxy geometry for a sphere. The viewer is assumed to be at position $(0,0)^\intercal$. The position of $g$ is determined by the constraint $g^\intercal(g-c)=0$. With $g$ known, proxy geometry, e.g. a rectangle, can be positioned and scaled to fully cover the area enclosed by $g$ and $g'$. $g$ can be moved by scaling it with a factor $\lambda \in \mathbb{R}_{>\not<}$, for example to position the proxy geometry in front of the sphere or at the center.

the origin $(0,0)^\intercal$. Because the position of $g$ is rotationally invariant with respect to the center of the sphere, it suffices to solve the problem in two dimensions. Let

$$c = \begin{pmatrix} c_x \\ 0 \end{pmatrix},\ g = \begin{pmatrix} c_x - \epsilon \\ h(\epsilon) \end{pmatrix}$$

be the center of the sphere and the wanted point, respectively. Using the Pythagorean theorem we calculate the height $h(\epsilon)$ of $g$ to be

$$h(\epsilon) = \sqrt{r^2 - \epsilon^2}.$$

From the fact that the direction of $g$ is perpendicular to the direction $(g-c)$ it follows that

$$\begin{aligned} g^\intercal(g-c) = 0 \\ &\Rightarrow \begin{pmatrix} c_x - \epsilon \\ h(\epsilon) \end{pmatrix}^\intercal \begin{pmatrix} -\epsilon \\ h(\epsilon) \end{pmatrix} = 0 \\ &\Rightarrow \epsilon^2 - c_x \cdot \epsilon + h(\epsilon)^2 = 0 \\ &\Rightarrow \epsilon^2 - c_x \cdot \epsilon + (r^2 - \epsilon^2) = 0 \\ &\Rightarrow \frac{r^2}{c_x} = \epsilon. \end{aligned}$$

And therefore $g$ has the form

$$g = \begin{pmatrix} g_x \\ g_y \end{pmatrix} = \begin{pmatrix} c_x - \frac{r^2}{c_x} \\ \sqrt{r^2 - (\frac{r^2}{c_x})^2} \end{pmatrix} = \frac{1}{c_x} \begin{pmatrix} c_x^2 - r^2 \\ r\sqrt{c_x^2 - r^2} \end{pmatrix}. \tag{2.1}$$

This yields both the distance to the viewer (given by $g_x$), as well as the minimal extent for the proxy geometry to fully cover the sphere as seen by the viewer (given by $g_y$). By scaling $g$ with a factor $\lambda \in \mathbb{R}_{>0}$ one can compute the extent $\lambda g_y$ for a proxy geometry with distance $\lambda g_x$ to the camera. For example, choosing $\lambda = c_x/g_x$ places the proxy geometry at the center of the sphere. Choosing $\lambda = (c_x - r)/g_x$ places the proxy geometry right in front of the sphere. Since the formula is derived for a viewer positioned at the origin $(0, 0, 0)^\intercal$, the positioning of the proxy geometry needs to be done in camera space. From the particle's position in camera space an orthogonal coordinate system can be derived, in which one coordinate axis is aligned with the direction vector pointing at the sphere's center. Within that coordinate system induced by the sphere's center, the proxy geometry is aligned to face the viewer and scaled according to $g_y$ to fully cover the sphere's screen-footprint.

### 2.1.2 OpenGL Implementation

When rendering particles as spheres, an OpenGL implementation of the sphere rendering algorithm needs to provide three basic components:

1. Supply particle data to OpenGL.
2. Correctly set up and draw proxy geometry.
3. Ray cast the sphere's geometry to determine discarded and non-discarded fragments. Write the latter to the framebuffer with a correct depth value.

Modern OpenGL offers several ways of implementing the different parts of the algorithm. The goal of this section is to analyze the different possible implementations and to determine what performance characteristics they have. A similar analysis has recently been done by [FGKR16]. Compared to that analysis, I will go into more depth explaining possible different low-level OpenGL implementation details and their semantics. I will focus on the impact of those implementation details in scenarios when lots of particles are rendered at the same time, meaning that a lot of spheres are going to overdraw each other. At the time of writing, the most recent OpenGL version is $4.5$, of which the *core profile* will be used for implementation considerations. Furthermore, only standard-conforming behavior will be used, which means that only OpenGL constructs are used which behave the same on all conforming OpenGL GPUs and drivers.

Given a list (i.e. an array) of particles $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^3 \times \mathbb{R}_{>0}$ consisting of a position and radius, the following section describes how to render those particles as spheres in OpenGL.

### Providing Data

Before rendering anything, the particle data needs to be communicated to OpenGL. Providing data to OpenGL is done via *Buffer Objects* or *Texture Objects*. Because *Texture Objects* are optimized to handle mostly image data, *Buffer Objects* are the natural choice for providing OpenGL with input like particle data. *Buffer Objects* [Wik17a] function as a piece of memory that is owned by OpenGL. The user can transfer data to the memory or use the memory in shaders by issuing draw calls.

*Buffer Objects* can be used in a variety of ways, but for drawing particles there are only two useful applications: using them either as a *Vertex Buffer* or as a *Shader Storage Buffer*[2]. *Vertex Buffers* provide data for individual vertices which result from a draw call. For drawing particles, this implies that each particle needs to be drawn as an individual vertex, leading to a single vertex shader invocation per particle which will be fed with input data for exactly that particle. The vertex input data is only available in the vertex shader and needs to be passed explicitly between shader stages if it is going to be used in a different stage.

Using *Buffer Objects* as *Shader Storage Buffers* makes the content of the buffer visible to any shader stage that declares such a buffer as input. The interpretation of the content of the buffer is under control of the shader invoked by a draw call. Generally, any combination of basic GLSL data and vector types, `struct` and array constructs of those types can be passed to shaders via *Shader Storage Buffers*. For particle rendering this implies that a C array of particles in a form like `float[4][n]` (holding a three-dimensional position and the radius) can be directly made visible to all shader invocations as an array `vec4[n]`, where `vec4` is the GLSL floating point vector type with four components. However, because each shader invocation (each individual vertex shader invocation) sees the whole buffer content, some sort of synchronization must take place during computations, such that each shader can decide which part of the data it should access. This synchronization can be done inside the vertex shader using

[2] All other possible use-cases for Buffer Objects use them as data sources for other mechanisms in OpenGL (e.g. as buffers for indexed rendering, as source for indirectly dispatched draw calls, or as memory for shader uniform updates.), but providing shaders with almost arbitrarily large particle data is only possible with Vertex Buffers, Shader Storage Buffers or Textures. Out of those, the Vertex Buffer and Shader Storage Buffer mechanisms allow for most flexibility regarding composition and structure of particle data. See [Wik17a] for Buffer Object use cases.

the built in GLSL variables `gl_VertexID` or `gl_InstanceID`, which provide the number of the current vertex inside the drawn primitive and the number of the current instance out of all drawn instances, respectively. For fragment shaders, no such enumeration of individual shader invocations is available. [3]

Allocating memory and writing data to *Buffer Objects* can be done using one of the two OpenGL functions `glBufferData` or `glBufferStorage`. The memory allocated by those functions differs in its properties and permissible use cases, and resulting from this there might be performance differences for the two memory types. `glBufferData` allocates *mutable* memory for a *Buffer Object* with a certain size each time it is called. This way the amount of memory held by the buffer can be changed at any time by reallocating the buffer's memory. In contrast, `glBufferStorage` allocates a fixed amount of memory as *immutable storage*, which means that the content of the buffer may be altered, but the *size* of the memory can not be changed after the initial allocation, so no reallocation is possible. While mutable memory gives the user more freedom by making reallocation possible at any time, immutable storage gives the OpenGL driver more freedom to optimize the usage of a buffer which will not change size. Both types of memory can be filled with data by instructing OpenGL to copy data into them (via `glBufferSubData`), or by requesting a pointer to the underlying memory (called *mapping the memory*) and writing directly to it. Mapped *mutable* memory must be unmapped again before it is used by OpenGL operations (e.g. draw calls), whereas *immutable* memory can be mapped *persistently*, i.e. the user can hold the pointer to the memory during OpenGL operations, but he is responsible to synchronize memory writes with OpenGL operations himself.

### Setting up geometry

With the particle data residing in OpenGL memory, the next step is to draw proxy geometry using that data. The remainder of this work will use a quad mesh as proxy geometry, which will be positioned and scaled according to Equation (2.1). Other primitives like regular convex polygons or the bounding box of the desired sphere might also be used as proxy geometry, but when drawing many particles the amount of vertices processed by OpenGL for each particle will significantly impact performance. Therefore, the amount of vertices used for proxy geometry should be minimized.

The final goal of the geometry stage is to draw geometry such that the resulting rasterized screen footprint covers the area where the particles sphere glyph will be seen by

[3] Some sort of enumeration for fragment shader invocations could be calculated from data like `gl_FragCoord`, the window position of the fragment. But this would not necessarily uniquely identify individual fragments resulting from a draw call, as multiple fragments might be assigned the same window position.

the viewer. This does not necessarily imply drawing an actual polygon mesh. The *NGSphereRenderer* module of the MegaMol™[GKM+15] framework draws point primitives (`GL_POINTS`) and scales their screen-space size according to the sphere's screen-space footprint. This technique processes only one vertex per particle, as compared to processing four vertices forming a quad. However, drawing `GL_POINTS` of arbitrary size is not guaranteed by the OpenGL specification and therefore not supported on all OpenGL-capable GPUs. So the method offering most portability will be explicit rendering of some triangle mesh, like a quad.

Given a *Buffer Object* filled with particle data, the *Buffer Object* can either be used as a *Vertex Buffer* (*VBO*) or as a *Shader Storage Buffer* (*SSBO*). The data of the buffer is going to be used in the vertex shader either as shader input for vertex attributes sourced from a *VBO*, or as a lookup into an array of particles provided by a *SSBO*. A vertex shader does not always need to have vertex attributes as input. In fact it is perfectly fine for a vertex shader to have no inputs in form of vertex attributes at all, as long as it writes something to the shader output variable `gl_Position`. In that case no data will be read from any *VBO* and it is permissible to use an empty *VBO* [4] before issuing a draw command.

One can render a quad for each particle in one of four ways:

1. Using a *VBO* as particle data source, draw one vertex per particle. The geometry shader stage unfolds the initial vertex into a quad and transforms it according to Equation (2.1).
2. Using a *SSBO* as particle data source, draw one vertex per particle, without vertex attribute inputs in the vertex shader and with an empty *VBO*. Based on the `gl_VertexID`, the vertex shader fetches particle data from the *SSBO* particle array. The geometry shader stage unfolds the initial vertex into a quad and transforms it accordingly.
3. Using a *SSBO* as particle data source, for each particle draw an instance of a quad which is sourced from a separate *VBO* containing four vertices as vertex attribute inputs to the vertex shader. Based on the `gl_InstanceID`, the vertex shader fetches particle data from the *SSBO* particle array and transforms the input quad vertex accordingly.
4. Using a *SSBO* as particle data source, for each particle draw an instance of a quad (four vertices), but the vertex shader is left without input attributes. Based on the `gl_InstanceID`, the vertex shader fetches particle data from the *SSBO* particle array. Based on the `gl_VertexID`, the vertex shader decides which corner of the

[4] As long as a valid *Vertex Array Object* is bound which is associated with that empty *VBO*.

quad is currently processed. Using the fetched particle data, the deduced vertex of the quad is transformed accordingly.

While method 1 and 2 use the geometry shader to unfold an initial vertex into a quad, method 3 and 4 directly dispatch vertices of a quad and transform them in the vertex shader. Method 2 and 4 take no vertex shader inputs besides the *SSBO*'s particle data buffer. The different combinations of shader inputs and active shader stages might lead to different performance characteristics of the implementations, which is the reason for evaluating all of them.

### Fragment operations

After the drawn proxy geometry has been rasterized, the fragment shader needs to ensure that the rendering of a sphere is correctly written to the framebuffer. Because the proxy geometry encloses the sphere's outline conservatively, fragments not participating in the appearance of the sphere must not alter the framebuffer.

Determining which fragment participates in rendering the sphere is done by testing a primary view-ray for intersection with the sphere. The view-ray is determined by the position of the viewer and the fragment's projected position in world space. Fragment shaders in OpenGL support perspectively-correct linear interpolation of arbitrary vertex data, which can be used to provide the fragment with its own 3d-position in world-space as sampled from the rasterized geometry. This way no tedious, handmade, back-projection of a fragment's window-coordinates into world coordinates is required. The view-ray can then be directly assembled from the cameras position and the fragment's interpolated world-position.

However, instead of performing ray-sphere intersection in world space, I argue that performing the intersection calculations in view space brings a few advantages. First, in view space the camera is at position $(0, 0, 0)^\intercal$, which makes setup of the ray and further calculations easier. Secondly, the spheres are rendered with respect to the camera positions anyway, because the proxy geometry has already been transformed by the view matrix. In large data sets, the distances between viewer and individual particles may become very large, which implies large floating point values possibly several magnitudes apart from each other. Such large differences in floating point values can lead to numerical errors. In view space, the sphere's positions are normalized with respect to the cameras position and the size of the view frustum, which will lead to less numerical errors in representing ray directions and when evaluating ray-sphere intersections.

Let $s$ be a sphere $s = (c, r) \in \mathbb{R}^3 \times \mathbb{R}_{>0}$ with center $c$ and radius $r$, then a point $x$ is on the sphere's surface iff $|c - x| = r^2$. Let $r = (\omega, d) \in \mathbb{R}^3 \times \mathbb{R}^3$ be a ray with the origin

$\omega$ and the direction vector $d \neq (0,0,0)^\intercal$, and let $p(t) = \omega + t \cdot d$ be a point on that ray parametrized by $t \in \mathbb{R}$.

Ray-sphere intersection is computed by plugging in the ray into the implicit sphere-surface equation

$$r^2 = |c - p(t)| = |c - \omega - t \cdot d| = (c - \omega - t \cdot d)^\intercal (c - \omega - t \cdot d)$$

which leads to a quadratic equation. Solving the equation for $t$ leads to a solution $t^*$ if the ray intersects the sphere. Defining $\gamma = (c - \omega)$, the solution is given by

$$t^* = \frac{\gamma^\intercal d - \sqrt{(\gamma^\intercal d)^2 - d^\intercal d(\gamma^\intercal \gamma - r^2)}}{d^\intercal d} \tag{2.2}$$

iff the radicand of the square root is positive.

If the ray hits the sphere, the intersection point $p = p(t^*)$ is projected into normalized device coordinates by the fragment shader and transformed according to the OpenGL viewport-transformation [Mar17, p. 438ff] to compute a depth value which can be written to the depth buffer by overwriting `gl_FragDepth`. Writing the correct depth value at sphere intersections is important to get a correct rendering when multiple spheres occlude each other. As already mentioned, fragments for which the ray does not intersect the sphere are discarded and do not contribute to the rendering.

Modern graphics hardware implements a series of optimizations, I will call them Early Depth Test, which minimize depth buffer writes and prevent fragments from being unnecessarily created [Wik15b]. Writing a fragment's depth value manually or discarding it turns off these optimizations because the rasterization pipeline can not assume certain properties anymore. When a fragment is discarded manually, more fragments must be generated which might have been occluded by the original one. When a fragment's depth value is written manually, the rasterization pipeline must assume that during rendering, any fragment will write an arbitrary depth value. Therefore, instead of z-culling some or most fragments, for all fragments a fragment shader must be executed which computes and writes the final depth value.

By guaranteeing that manual depth value writes will only *increase* with respect to the original `gl_FragDepth` value, a fragment shader can give the rasterization pipeline an opportunity to optimize fragment processing even if a depth value is manually written. To implement this, the proxy geometry needs to be drawn *in front of* the spheres, and the fragment shader must declare explicitly that depth writes will only increase the initial depth value of the fragment. This is done by declaring the fragment's depth value using the following statement in shader code: `layout (depth_greater) out float gl_FragDepth;`. Notifying the OpenGL implementation of such rendering intention could help optimize situations where lots of overdraw occurs, which is the case when millions of particles are rendered at once.

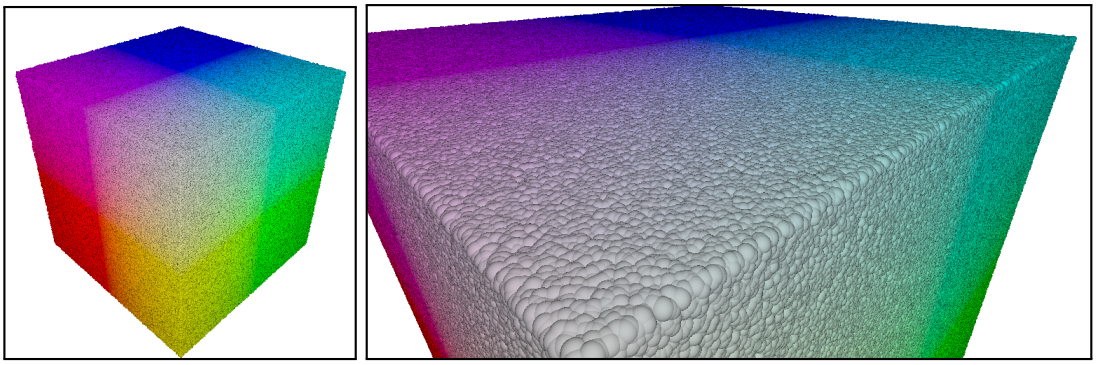


**Figure 2.3:** Particle data sets used for evaluating performance of the APU, TITAN and RadeonR9 systems. Left: Overview of 10 million randomly positioned particles in a $[-1, 1]^3$ cube with radius $0.01$. Right: Closeup of same cubical setup, but of a data set with 50 million particles. The overdraw of overlapping spheres stresses fragment processing, because it is not known beforehand which fragment may be discarded or may survive, writing an arbitrary depth value to the depth buffer.

### 2.1.3 Performance Measurements and Implications

Combining different OpenGL mechanisms to allocate memory, setup geometry and perform fragment shader operations results in lots of possible implementations. The following implementation choices are available for each of the three aspects (memory allocation, geometry setup, fragment operations):

1. Memory Allocation
   - **Data**. Allocate mutable memory using `glBufferData`.
   - **Storage**. Allocate immutable memory using `glBufferStorage`.
2. Geometry Setup
   - **VBO + GS**. Bind memory as VBO and unfold an initial single vertex inside the geometry shader into quad. Draw using `glDrawArrays`.
   - **SSBO + GS**. Bind memory as SSBO and unfold an initial single vertex inside the geometry shader into quad. Draw using `glDrawArrays`.
   - **SSBO + VS Empty Quad**. Bind memory as SSBO and draw an instance of a quad per particle. Vertex shader inputs stay empty. Draw using `glDrawArraysInstanced`.

- **SSBO + VS Quad**. Bind memory as SSBO and draw an instance of a quad per particle. Vertex shader inputs are corners of a quad. Draw using `glDrawArraysInstanced`.

3. Fragment Operations
   - **Point**. Draw only `GL_POINTS`. Only one fragment gets drawn per particle.
   - **Quad**. Draw the proxy-geometry quad without discarding fragments or writing depth.
   - **Sphere**. Draw a full sphere. Discard fragments and write depth, but write depth arbitrarily without optimizing depth writes.
   - **Sphere Early**. Draw a full sphere, but optimize depth writes by guaranteeing to OpenGL only increasing depth values. This allows the pipeline to use early z-culling for fragments.

To benchmark the possible implementation choices, a highly configurable rendering application has been implemented which allows to combine these different options arbitrarily. In the following, performance measurements for the different implementation combinations are reported.

Additionally, different methods of updating the OpenGL Buffer Object's contents are available. This is interesting for dynamic data sets, where regular updates of memory contents are necessary. Possible options to update OpenGL Buffer Object contents depend on whether *mutable* Data memory or *immutable* Storage memory has been allocated.

1. **Data**
   - **Buffer**. Reallocate the whole buffer and initialize it with data using `glBufferData`.
   - **SubBuffer**. Copy data to the buffer using `glBufferSubData`.
   - **Orphan**. Call `glBufferData` with a `nullptr` as source buffer to signal to OpenGL that new memory should be allocated for the Buffer Object. OpenGL will correctly handle lifetime and synchronization of the old memory while it is used by OpenGL commands. Then upload the actual data update using `glBufferSubData`.
   - **Map**. Map a pointer to the buffer's memory, copy particle data to the memory using that pointer, un-map the pointer before drawing.
2. **Storage**

- **Persistently Map**. Permanently map a pointer to the buffer's Storage memory and copy particle data using that pointer. Explicitly notify OpenGL of the changed buffer contents by *flushing* the changes. For immutable memory, the mapped pointer can stay mapped during other OpenGL operations.

Immutable storage could also be updated using the *SubBuffer* and *Map* technique, but *Persistently Mapping* the memory allows for a wider range of use cases, since the mapped pointer can be written by other threads and does not need to be un-mapped before the memory contents are used by OpenGL. While the *Buffer* and *SubBuffer* methods depend on the internal synchronization of memory accesses by OpenGL, the *Orphan* technique tries to avoid waiting for operations on a buffer to finish by re-allocating the underlying memory. The *Map* and *Persistently Map* methods need specific synchronization by the user to ensure that the memory is not used by OpenGL commands anymore before updating the contents. This is done by issuing OpenGL *Sync Objects* after draw calls, which are signaled by the OpenGL pipeline after the draw call finished. After the Sync Object has been signaled, the memory used by draw commands can be updated. Resources on such OpenGL techniques are available in the official OpenGL Wiki and documentation [Wik15a]. A good overview and description of techniques is also given by [HM12]. No performance guarantees about any behaviors are made by OpenGL. The OpenGL driver and hardware are free to optimize for different techniques and use cases, which is why different GPUs, different drivers or driver versions, and GPUs of different hardware vendors might show completely different performance characteristics.

Performance measurements have been done on three different systems.

- The **APU** system is a laptop with Arch Linux (Kernel 4.11.7), AMD A4-5000 4-core processor @ 1.5GHz and integrated Radeon HD 8330 graphics chip running the open source Mesa 17.1.4 *amdgpu* OpenGL driver officially supported by AMD. The compiler used is the GCC version 7.11 with maximum optimization level, link-time optimization and the C++14 standard enabled.
- The **TITAN** system is a Windows 10 desktop PC with Intel Core i7-3820 4-core processor @ 3.6GHz and a GeForce GTX TITAN graphics card running the official NVIDIA driver version 376.53. The program is compiled on Visual Studio Enterprise 2015 in Release mode.
- The **RadeonR9** system is a Windows 10 PC with Intel Xeon E5-2630 v4 8-core processor @ 2.2GHz and an AMD Radeon R9 200 Series graphics card running the official Radeon driver version 17.10. The program is compiled on Visual Studio Enterprise 2015 in Release mode.

The measured scenarios are the rendering performance of different sphere rendering configurations for memory allocation, geometry setup and fragment operations, as well as data streaming to the GPU. The sphere rendering is tested on a synthetic **Cube** data

set containing 10 million particles with radius $r = 0.01$ which are equally distributed in a $[-1, 1]^3$ domain. Figure 2.3 shows images of such Cube data sets. The buffer data streaming is tested for the memory update methods described above, with the *Cube* data setup for varying particle amounts of $500,000$, 1 million, 2 million, 5 million and 10 million particles drawn off-screen as points. All times are given as measured milliseconds per frame. As a reference to interpret the measurements, for a target frame rate (frames per second) of 60fps, a frame has roughly 16.6 milliseconds to render. For a frame rate of 30fps and 10fps, a frame has 33 milliseconds and 100 milliseconds to render, respectively.

Frame times are averaged over a series of ten consecutive frames. This way, outliers and inconsistencies in performance are easy to identify. During measurements, such outliers were ignored and frame times best representing the overall performance were noted down. For some configurations, the rendering would not show a stable frame time, but instead frame times would periodically jump between values in some range (for example, in a range of $67$, $69$, $72$, $75$.). For such scenarios, the mean value of the time values in that range is reported. Although this does not represent the unstable nature of the measured times, the resulting mean values allow for cleaner presentation of the measurements. Where different rendering methods showed similar but unstable performance, the resulting mean values turned out to represent the relative performance of the methods quite well. Moreover, the reported frame times for the examined systems should not be considered as absolute performance values. The measurements of different rendering configurations should be interpreted relative to the other possible configurations on the same GPU. Performance measurements of one and the same method would sometimes differ within a few milliseconds (e.g. after restarting the computer), presumably depending on internal GPU driver metrics or other GPU states influenced by the operating system. In such cases however, all measured methods showed a shift in frame times which resulted in consistent relative performance of the methods. Times for rendering modes showing a stable performance were rounded to one decimal place. Times given without a decimal place represent mean values for a range of measured times. No other GPU-intensive tasks were running on the systems during measurements

Measurements for the *APU* system are presented in Table 2.1 and Table 2.2. On the APU system, the memory types Data and Storage both consistently gave identical results in all measurements. The measured times for different render method combinations in Table 2.1 show that drawing Points is equally fast across all methods, with the VBO method being a little faster than the others. When rendering Quads or Spheres, the geometry shader based *VBO/SSBO+GS* methods seem to impose a heavy load on geometry processing (see subscripts in the table for geometry processing times). Activating early depth tests by ensuring only increasing depth writes significantly improves performance on all methods. The early depth test reduces fragment processing for spheres so far

| Memory: **Data & Storage** | Point | Quad | Sphere | Sphere Early |
|---|---|---|---|---|
| VBO+GS | $174_{23}$ | $855_{807}$ | $2000_{1500}$ | $1563_{1508}$ |
| SSBO+GS | $177_{23}$ | $858_{807}$ | $2028_{1500}$ | $1566_{1500}$ |
| SSBO+VS Empty Quad | $177_{23}$ | $138_{117}$ | $500_{150}$ | $192_{150}$ |
| SSBO+VS Quad | $177_{23}$ | $157_{150}$ | $500_{136}$ | $\mathbf{184_{136}}$ |

**Table 2.1:** Different rendering methods for spheres on the **APU** system. Times in milliseconds per frame. The numbers represent times for the memory types Data and Storage, which gave identical results. 10 million spheres in a cubical domain are drawn with a $512^2$ viewport, with all spheres being inside the view frustum. The subscript numbers denote times for pure geometry-processing when no sphere is inside the camera frustum and no fragments get created. Fastest method in bold.

| Points, VBO & SSBO | 500K | 1M | 2M | 5M | 10M |
|---|---|---|---|---|---|
| Static Draw | 1.7 | 2.9 | 5.3 | 12 | 23 |
| Data+Buffer | 5.5 | 9.7 | 19 | 45 | 110 |
| Data+SubBuffer | 5.5 | 9.7 | 19 | 45 | 110 |
| Data+Orphan | 5.5 | 9.7 | 19 | 45 | 110 |
| Data+Map | 5.5 | 9.7 | 19 | 45 | 89 |
| Storage+Pers.Map | 5.5 | 9.7 | 19 | 45 | **88** |

**Table 2.2:** Different data streaming modes for different amounts of updated particles on the **APU** system. Times in milliseconds per frame. Times are for point-rendering, the VBO and SSBO render methods all gave identical results. Different amounts of points were rendered outside the camera view frustum to measure only geometry processing overhead and data update times per frame. The *Static Draw* times represent the baseline for rendering geometry without data update overhead. The other modes are combinations of Buffer Object memory allocation and update variants. Fastest method in bold.

that the geometry processing becomes largely the bottleneck of the rendering. All in all, for the APU system the *SSBO+VS Quad* vertex shader based geometry setup with early depth culling activated is the best method to render spheres. The memory streaming measurements in Table 2.1 show similar results across most data set sizes and update mechanisms. The data streaming measurements in Table 2.2 show that for large data sets of 10 million particles (roughly 152MiB), the memory mapping methods for Data and Storage memory become faster than the other variants. However, rendering so many particles at a time is already non-interactive on the APU system, at roughly 5.5fps.

| Memory: **Data** | Point | Quad | Sphere | Sphere Early |
|---|---|---|---|---|
| VBO+GS | $6.65_{2.9}$ | $13.4_{9.0}$ | $25.8_{13.8}$ | $25.5_{13.8}$ |
| SSBO+GS | $7.05_{3.0}$ | $14.5_{9.2}$ | $25.6_{13.7}$ | $\mathbf{25.4_{13.7}}$ |
| SSBO+VS Empty Quad | $52.0_{49.6}$ | $53.7_{49.6}$ | $59.5_{51.5}$ | $59.5_{51.5}$ |
| SSBO+VS Quad | $52.2_{49.6}$ | $53.7_{49.5}$ | $59.4_{50.9}$ | $59.4_{50.8}$ |
| Memory: **Storage** | Point | Quad | Sphere | Sphere Early |
| VBO+GS | $28.7_{28.7}$ | $37.4_{33.7}$ | $49.2_{38.4}$ | $49.1_{38.4}$ |
| SSBO+GS | $28.8_{28.7}$ | $36.9_{33.3}$ | $\mathbf{48.3_{38}}$ | $48.8_{38}$ |
| SSBO+VS Empty Quad | $78_{78}$ | $78_{78}$ | $81_{78}$ | $81_{79}$ |
| SSBO+VS Quad | $78_{78}$ | $79_{80}$ | $82_{80}$ | $82_{80}$ |

**Table 2.3:** Different rendering methods for spheres on the **TITAN** system. Times in milliseconds per frame. The numbers represent times for the memory types Data (upper table) and Storage (lower table). 10 million spheres in a cubical domain are drawn with a $1024^2$ viewport, with all spheres being inside the view frustum. The subscript numbers denote times for pure geometry-processing when no sphere is inside the camera frustum and no fragments get created. Fastest method in bold.

Measurements for the TITAN system are shown in Table 2.3, Table 2.4 and Table 2.5. The sphere rendering methods in Table 2.3 show results for rendering with *Data* and *Storage* memory, which show significant differences in performance. The Data memory type is faster in all aspects of the comparison. The Storage memory increases geometry-processing costs substantially. This could be explained by driver optimizations or better hardware capabilities regarding the *Data* memory type. Contrary to the APU system, the TITAN system performs better on rendering methods utilizing the geometry shader (VBO+GS and SSBO+GS), on both memory types. The larger amount of dispatched vertices or the instanced rendering used for the SSBO+VS methods seem to be more demanding. As the off-screen geometry processing times (subscript numbers) for the *SSBO+VS* modes show, geometry load makes up almost all of the processing time. This may be a hint that other factors also measured by the off-screen geometry processing time, like the OpenGL pipeline state resulting from instanced rendering, influence the performance of the *SSBO+VS* modes in a negative way.

Also very interesting is the fact that the Sphere Early method does not profit from the early depth test, it is only slightly faster than the normal Sphere method. The fragment processing times for both Sphere rendering methods are roughly $10$ms across all geometry setup variants (after subtracting the geometry processing time), which indicates that the rendering performance is highly dependent on the processing of

| Stream Points via VBO | 500K | 1M | 2M | 5M | 10M |
|---|---|---|---|---|---|
| Static Draw | 0.2 | 0.3 | 0.6 | 1.4 | 2.8 |
| Data+Buffer | 1.7 | 3.4 | 6.7 | 16.3 | **32.4** |
| Data+SubBuffer | 1.71 | 3.4 | 5.7 | 16.3 | **32.4** |
| Data+Orphan | 1.7 | 4.3 | 6.7 | 16.3 | 32.4 |
| Data+Map | 2.9 | 5.8 | 11.5 | 28.8 | 56 |
| Storage+Pers.Map | 2.9(1.5) | 5.9(2.9) | 11.5(5.8) | 28.7(14) | 56(28) |

| Stream Points via SSBO | 500K | 1M | 2M | 5M | 10M |
|---|---|---|---|---|---|
| Static Draw | 0.2 | 0.3 | 0.6 | 1.4 | 2.8 |
| Data+Buffer | 5.5 | 11.9 | 23.3 | 58 | 125 |
| Data+SubBuffer | 1.7 | 3.4 | 6.6 | 16.3 | **32** |
| Data+Orphan | 5.5 | 11.9 | 23.3 | 57.7 | 125 |
| Data+Map | 2.9 | 6.0 | 11.7 | 28.4 | 56.6 |
| Storage+Pers.Map | 2.8(1.5) | 5.8(2.9) | 11.5(5.8) | 28.5(14.4) | 56.6(28.7) |

**Table 2.4:** Different data streaming modes for different amounts of updated particles on the **TITAN** system. Times in milliseconds per frame. Times are for point-rendering sourced from a VBO (upper table) and SSBO (lower table). Different amounts of points were rendered outside the camera view frustum to measure only geometry processing overhead and data update times per frame. The *Static Draw* times represent the baseline for rendering geometry without data update overhead from *Data* memory. The numbers in brackets are static draw times for *Storage* memory. The other modes are combinations of Buffer Object memory allocation and update variants. Fastest method in bold.

vertices and the memory type they are sourced from. Using the SSBO as particle source seems to be slightly faster, but other factors have much more impact on performance. The impact of geometry processing can also be seen in Table 2.5, which shows rendering times for different sizes of particle data sets, where geometry processing consistently takes about two thirds of the whole rendering time. So on the TITAN system, using geometry shaders for proxy geometry setup, with the particle data coming from a *Data* memory type, and allowing early depth tests is the most efficient method of drawing spheres.

The data streaming measurements in Table 2.4 suggest that using the OpenGL `glBufferData` and `glBufferSubData` functions are the fastest ways to update OpenGL memory on the TITAN system. Although mapping pointers to OpenGL memory and performing the data copy operation manually is said to be faster [HM12], the measured

| Memory Type | Data Set Size | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 10M | 15M | 20M | 25M | 30M | 35M | 40M | 45M | 50M |
| Data | $25.4_{13.7}$ | $38.7_{21}$ | $45.3_{31.9}$ | $57.2_{40}$ | $64_{43}$ | $73.2_{49.3}$ | $83_{55}$ | $93_{62}$ | $113_{70}$ |
| Storage | $48.8_{38}$ | $68_{58}$ | $87_{76}$ | $117_{103}$ | $140_{123}$ | $162_{144}$ | $186_{164}$ | $208_{185}$ | $232_{205}$ |

**Table 2.5:** Times in milliseconds per frame for rendering particle data of size 10 million up to 50 million in the *Cube* scenario, with all particles inside the view frustum. The rendering uses the fastest combination of methods for the **TITAN** GPU: geometry is set up via SSBO+GS and fragment operations are done with the Sphere Early method. The subscript numbers denote times for pure geometry-processing when no sphere is inside the camera frustum and no fragments get created. Throughout all data set sizes, geometry-processing takes up roughly two thirds of the whole processing time.

frame times in context of particle rendering favor the Buffer and SubBuffer methods. The Orphan method, which aims at reducing synchronization overhead for the OpenGL driver, shows no benefit over the other methods. One interesting observation is that the frame time for the mapped Storage memory is the same as for mapped Data memory, but the base workload for *Static Draw* for the two memory types differs significantly. The Static Draw frame time is measured for rendering particles outside the view frustum with static memory contents, i.e. a non-streaming scenario meant for comparison with the streaming methods. Subtracting this Static Draw time from streaming times, in assumption of thereby removing the overhead for drawing geometry at all and ending up with pure data upload time, Storage memory seems to have faster *data upload* times at the cost of slower rendering, while Data memory has the reverse effect. However, although mapped Storage memory is not the fastest data-upload method, permanently mapping pointers of OpenGL Storage memory is an interesting possibility when designing rendering algorithms.

Measurements for the RadeonR9 system are shown in Table 2.6, Table 2.7 and Table 2.8. The sphere rendering methods in Table 2.6 show no significant performance differences for the tested modes or memory types. While differences clearly exist between the different variants, the performance impact of choosing a non-optimal configuration for rendering is not as large as on the TITAN system. Big differences occur for point-rendering, where the GS methods are notably faster than the VS methods for Data type memory. This probably is due to OpenGL overhead resulting from the instanced rendering used for the VS methods. Storage memory seems to incur a general performance hit regarding geometry processing base times, which may be the reason why configurations using Storage memory are $1 - 7$ milliseconds slower when rendering the Quad, Sphere and Sphere Early methods. However, for data streaming (see Table 2.7), persistently mapped Storage memory can update data almost as fast (with $39.6$ms per frame) as the

| Memory: **Data** | Point | Quad | Sphere | Sphere Early |
|---|---|---|---|---|
| VBO+GS | $6.6_{3.5}$ | $29.8_{28.3}$ | $61_{28.3}$ | $\mathbf{32_{28.3}}$ |
| SSBO+GS | $6.5_{3.5}$ | $30_{29}$ | $63_{29}$ | $32_{29}$ |
| SSBO+VS Empty Quad | $13.3_{13}$ | $32_{31.9}$ | $65_{31.9}$ | $33_{32}$ |
| SSBO+VS Quad | $13.3_{13}$ | $32.4_{32}$ | $65_{32}$ | $34_{32}$ |

| Memory: **Storage** | Point | Quad | Sphere | Sphere Early |
|---|---|---|---|---|
| VBO+GS | $19.1_{19.3}$ | $32_{34}$ | $66_{34}$ | $39_{34}$ |
| SSBO+GS | $19.1_{19.1}$ | $32_{34}$ | $65_{34}$ | $39_{34}$ |
| SSBO+VS Empty Quad | $19_{19.1}$ | $31_{34}$ | $63_{34}$ | $36_{34}$ |
| SSBO+VS Quad | $19_{19.1}$ | $32.5_{31.9}$ | $62_{32.5}$ | $\mathbf{33_{32.5}}$ |

**Table 2.6:** Different rendering methods for spheres on the **RadeonR9** system. Times in milliseconds per frame. The numbers represent times for the memory types Data (upper table) and Storage (lower table). 10 million spheres in a cubical domain are drawn with a $1024^2$ viewport, with all spheres being inside the view frustum. The subscript numbers denote times for pure geometry-processing when no sphere is inside the camera frustum and no fragments get created. Fastest method in bold.

| Stream Points via VBO | 500K | 1M | 2M | 5M | 10M |
|---|---|---|---|---|---|
| Static Draw | 0.3 | 0.4 | 0.9 | 1.9 | 3.2 |
| Data+Buffer | 2.9 | 5.3 | 10.4 | 28.1 | 52.8 |
| Data+SubBuffer | 1.9 | 3.9 | 7.8 | 19.5 | **37.7** |
| Data+Orphan | 2.9 | 5.3 | 10.4 | 28.2 | 52.8 |
| Data+Map | 2.4 | 4.4 | 8.5 | 22.5 | 40.7 |
| Storage+Pers.Map | 1.9(1.3) | 3.9(2.3) | 7.9(3.9) | 21.3(9.8) | 39.6(19.1) |

**Table 2.7:** Different data streaming modes for different amounts of updated particles on the **RadeonR9** system. Times in milliseconds per frame. Times are for point-rendering. Rendering with VBO and SSBO as particles sources gave identical results. Different amounts of points were rendered outside the camera view frustum to measure only geometry processing overhead and data update times per frame. The *Static Draw* times represent the baseline for rendering geometry without data update overhead from *Data* memory. The numbers in brackets are static draw times for *Storage* memory. The other modes are combinations of Buffer Object memory allocation and update variants. Fastest method in bold.

| Memory Type | Data Set Size | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 10M | 15M | 20M | 25M | 30M | 35M | 40M | 45M | 50M |
| Data | $29.3_{27.9}$ | $42.9_{41.4}$ | $56.3_{54.6}$ | $69.7_{68}$ | $89_{87}$ | $114_{111}$ | $137_{128}$ | $151_{147}$ | $167_{161}$ |
| Storage | $34_{32}$ | $59_{54}$ | $75_{73}$ | $90_{87}$ | $120_{115}$ | $140_{135}$ | $160_{155}$ | $177_{170}$ | $205_{194}$ |

**Table 2.8:** Times in milliseconds per frame for rendering particle data of size 10 million up to 50 million in the *Cube* scenario, with all particles inside the view frustum. The rendering uses the fastest combination of methods for the **RadeonR9** GPU: for the memory type Data, geometry is set up via VBO+GS; for memory type Storage, geometry is set up via SSBO+VS Quad. Fragment operations are done with the Sphere Early method. The subscript numbers denote times for pure geometry-processing when no sphere is inside the camera frustum and no fragments get created. Throughout all data set sizes, geometry-processing takes up almost all of the processing time.

fastest Data update method via SubBuffer ($37.7$ms per frame). As with the TITAN system, the base offset for geometry processing is larger for Storage memory than for Data memory. So Data memory seems to be optimized for rendering static, not re-uploaded data on both the TITAN and the RadeonR9 system. The Orphan buffer update technique shows no performance gain on the RadeonR9 system either, which makes me wonder whether I implemented it incorrectly, or whether none of the used OpenGL drivers implements this optimization. The measurements for rendering particle data of different sizes in Table 2.8 suggests that the RadeonR9 system is largely bottlenecked by the geometry processing, which takes roughly $95\%$ of the rendering time.

The conclusion drawn from these measurements is that there is no good way to know the impact of implementation choices on OpenGL rendering performance, except after explicitly measuring all possible variants. Even then, no general conclusions can be drawn from measurements on only one system. Depending on the graphics hardware, hardware vendor and OpenGL driver version, performance of rendering methods may drastically differ. For example, the assumption that the geometry shader stage introduces an overhead and thus leads to slower rendering is only valid on the APU system, while on the TITAN system the geometry shader variant has the best performance in all scenarios. However, the APU system has an integrated GPU chip which in general is not very fast and uses a fairly outdated GPU architecture from 2013, and as can be seen with the other two tested systems, geometry shaders on modern GPUs in fact seem to give the best performance for setting up particle geometry. On the RadeonR9 system, both geometry setup variants, either using the geometry shader or only the vertex shader, can be fast, depending on the memory type they read particle data from. Similar conclusions can be drawn for the fragment early depth test, the nature of the memory allocation method for buffers and buffer update mechanisms. While on the

NVIDIA TITAN system the early depth test has no significant influence on performance, the APU and RadeonR9 systems benefit greatly when early depth test is explicitly announced to the fragment shader. On all three systems there is no significant difference whether the geometry shader method GS is fed particle data via VBO or via SSBO. The memory type an OpenGL buffer is allocated from (either Data or Storage) however has a significant impact on rendering performance on the TITAN system, while on the RadeonR9 system performance differences are not as big. Moreover, it is not obvious to generalize OpenGL behavior from such measurements. For example, Storage memory may have the fastest update mechanism on all tested systems when the times for static rendering are subtracted from streamed rendering times, but it is not clear whether that calculation is correct, and whether the resulting buffer update time will reflect OpenGL behavior for other rendering workloads using Storage memory.

The results of the undertaken measurements show that on all examined systems, processing of geometry outweighs the processing times of fragments. When early z-culling of fragments is made possible to the GPU, fragment processing times of correctly rendered sphere glyphs are on par with simple quad rendering on the APU and RadeonR9 systems. On the TITAN system, activating an early z-culling of fragments does not improve rendering performance.

## 2.2 Ray Casting pkd-Trees on the GPU

Analyzing the sphere ray casting algorithm presented above, it is clear that for each drawn particle at least one vertex shader and potentially lots of fragment shaders are dispatched. The question arises whether there is a better, faster approach to render lots of spheres, an approach that avoids processing lots of vertices and that minimizes overdraw by creating only few fragments.

Since the sphere ray casting algorithm is just a way of setting up view-rays to test for intersection with a sphere object, a generalization of that approach towards a ray traversal of a general data structure on the GPU is desirable. Recently, the pkd-tree data structure has been introduced by [WKJ+15], which utilizes a kd-tree as spatial data structure to accelerate ray casting spheres on the CPU. In the context of this work, the pkd-tree has two important advantages compared to other spatial data structures: it does not need to store additional meta data and it is fairly simple to traverse. Spatial data structures, like kd-trees or bounding volume hierarchies used to accelerate ray tracing, must usually store meta data like links to tree nodes and links to contained geometry, which increases memory footprint of those data structures [HH11; VHB14; WSWG13]. When working with large data sets, it is undesirable to also store meta data that scales with the size of the data. Also, an evaluation of a data structure on the GPU needs to work under the constraints of the OpenGL computational model - either of the fragment shader stage, or the compute shader. In both shaders, recursion and pointers to arbitrary memory positions are not possible, and the pkd-tree data structure does not make use of those concepts.

When used for ray tracing geometry consisting of triangles, kd-trees divide the set of triangles into two parts by placing axis aligned splitting-planes. A splitting plane can generally be placed at an arbitrary position along a main axis, but the cost to traverse the tree can be minimized if splitting planes are placed according to the *surface area heuristic* [WH06]. In contrast, the original kd-tree data structure introduced by [Ben75] is defined for $k$-dimensional points in space, instead of triangles. Assuming that the point data is sorted along one of the main coordinate axes, the original kd-tree places the splitting plane *on* the median point of that sorting. Thus, the point data set is split into two equally sized partitions, which are recursively partitioned in the same manner. In the following, the definition of kd-trees for points will be used, and that data structure is utilized to ray trace particle spheres on the GPU according to the algorithm presented by [WKJ+15]. I will omit describing the original pkd-tree algorithm on the CPU, and will instead directly report a variant of the algorithm which I ported to the GPU for evaluation in GLSL fragment shaders. In the following, the terms kd-tree and pkd-tree will be used synonymously, with the slight difference that *kd-tree* refers to the memory

layout and sorting of particle data, and the *pkd-tree* additionally holds further meta data which is important for actual ray traversal of that structure.

### 2.2.1 Algorithm on the GPU

Given a list of particles $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^3 \times \mathbb{R}_{>0}$ as an array in contiguous memory, a kd-tree structure is built by resorting the array's elements. The properties of the kd-tree are then used to implement a ray traversal algorithm which determines the closest sphere in the tree that is intersected by the ray. The particle data and further pkd-tree meta data is prepared on the CPU and uploaded to the GPU for traversal (and thus particle rendering).

The view-ray for pkd-tree traversal is set up in the fragment shader after drawing a conservative axis-aligned bounding box of all particles (i.e. spheres) as proxy geometry. The pkd-tree structure containing sorted particle data is passed to the fragment shader using a *Shader Storage Buffer*. By traversing the pkd-tree with the view-ray, each fragment shader invocation determines the closest intersected sphere and writes proper depth and color values to the framebuffer, or discards the fragment if the ray does not intersect a sphere.

#### Sorting the Data

The central idea of the kd-tree is that the sorting of the data in memory implies a spatial sorting of objects in the scene. Traversing the scene with a ray then requires looking at few data entries in memory, leading to faster overall testing of intersections with scene objects. To keep the derivation simple, let the number of particles in the scene be $n = 2^k - 1$, and let no two particles have the same position. The number of particles ensures that after removing a particle from the list, the remaining particles can be split into two partitions of equal size. [5]

Let $(p_1, p_2, \ldots p_n)$ be an array of particles $p_i \in \mathbb{R}^3 \times \mathbb{R}_{>0}$. The kd-tree is built from the initial array using the particles' 3d-positions. This is the original method for kd-trees by [Ben75]. First, choose a split axis $\alpha$ as one of the three coordinate axes $\alpha \in \{x, y, z\}$. Sort the particles with respect to that dimension, particles with a smaller value in that dimension are sorted before (left to) particles with a larger value. Because $n$ is an odd number, there is an unambiguous median particle $p_m$ in that sorting. Choose that

[5] The particle number $n$ arises from solving the recursion $c_{i+1} = 2 \cdot c_i + 1$ (with $c_1 = 1$) for $c_{i+1}$. The recursive formula encodes the number of objects in a tree that is constructed from two smaller trees of equal size.

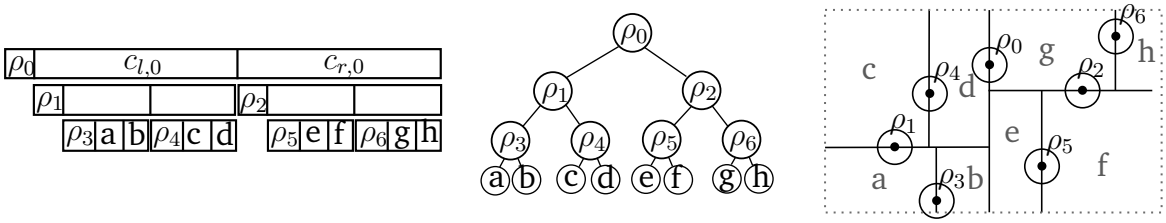



**Figure 2.4:** Memory layout of the kd-tree and the resulting spatial relations of particles. Left: Recursive sorting of particles in memory. The median value is written at the start of memory for a tree level, then the particles of the left and right subtrees follow. The subtrees themselves are again valid kd-trees. Middle: Binary tree which is encoded in the memory ordering. Right: Spatial sorting of particles resulting from the memory layout of the tree. Each particle defines a splitting plane through the data set. The leaf-nodes *a, b, c, d, e, f, g, h* show how further subtrees would correspond in memory, in the tree representation and in space.

median particle to be the root node $\rho = \{p_m\}$ of the kd-tree, and partition the rest of the particles into two groups: particles with smaller $\alpha$-value go into the left subtree $c_l = \{p \mid p \text{ smaller than } p_m\}$, particles with larger $\alpha$-value go into the right subtree $c_r = \{p \mid p \text{ greater than } p_m\}$. In the sorted array, the left and right subtree are already partitioned to be left and right of the median: $(c_l, \rho, c_r)$. For the sake of a simpler traversal algorithm we reorder the array by writing the root node at the start of the array: $(\rho, c_l, c_r)$. The left and right subtree (or *child trees*) have exactly $n' = 2^{k-1} - 1$ elements and the array's subranges containing the child trees can be recursively sorted to form kd-trees in the same manner. For simplification purposes, during recursive calls of kd-tree generation the sorting-axis $\alpha$ is chosen in the order $x, y, z$ repeatedly. The recursion stops when child trees have size $1$. Figure 2.4 depicts a kd-tree built in that manner, with the corresponding memory layout, the implicitly encoded binary tree and the resulting spatial sorting of particles. The resulting memory layout is called *depth-first tree layout* and allows to handle subsets of the data set as smaller kd-trees. The original pkd-tree algorithm by [WKJ+15] places child nodes of a parent node at memory position $i$ at memory positions $2i + 1$ and $2i + 2$, which leads to subtrees being interleaved in memory instead of being tightly packed together.

Since the kd-tree is encoded in the permutation of the original array, the data structure does not need additional meta information other than the size of the data set and the order of the axes the algorithm used to sort the data. Additionally, the children of a tree are themselves valid trees and are tightly packed in memory. This way, subtrees can be processed individually, e.g. uploaded to the GPU in small chunks, without the need to access the whole data array at once.

### Ray Traversal

Given a view-ray into the scene and a kd-tree as described above, the objective is to determine whether the ray intersects a sphere that is defined by a particle in the kd-tree, and if so, determining the closest sphere that is intersected. Figure 2.4 depicts the kd-tree structure induced by the memory ordering and the resulting spatial sorting of the particles. The basic idea is to first test the ray for intersection with the sphere at the root node of the tree, and then to traverse child nodes the ray passes through. Deciding whether the ray passes through space covered by a subtree is done by testing whether the ray intersects the subtree's axis-aligned bounding box. Because the kd-tree encodes axis-aligned slit planes that partition the data set at root nodes, bounding boxes for subtrees can be derived from bounding boxes of their parent nodes. Therefore, bounding boxes need not be explicitly stored, except for the initial bounding box of the whole kd-tree. That bounding box, along with the size of the tree, is the only meta data needed besides the sorted particles array. During traversal of subtrees, a hit value $t_{hit} \in \mathbb{R}_{\geq 0}$ is carried along, which stores the nearest intersection with a sphere the ray has seen so far. Subtrees with bounding box entry points $t > t_{hit}$ which lie behind the current nearest intersection need not be traversed.

There are two kinds of axis aligned bounding boxes in the pkd-tree. The *point bounding box* is defined by the 3d-positions of the particles, while the *extended bounding box* encloses the spheres defined by the particles. So there is a direct relationship between the point bounding box and the extended bounding box, where enlarging the former by the particle's radius yields the latter. Since each particle in the data set can have an individual radius, the maximum radius $r_{max}$ of all radii is used to derive the enlarged bounding box. This way, given some bounding box and the root node of a kd-tree, point and extended bounding boxes can be derived for the left and right child trees. If a traversing ray intersects the extended bounding box of a kd-tree, it might intersect spheres in that tree. If it does not intersect the bounding box, it will not intersect any sphere in that tree. This easy availability of subtree-bounding-boxes is an important observation of the original pkd-tree method by [WKJ+15]. The bounding box intersection test can be efficiently implemented using the DDA-based method presented by [WBMS05]. The resulting central idea of the algorithm is to traverse the tree and prune subtrees which are not intersected by the ray. The sphere at each subtree's root node is tested for intersection with the ray, which realizes the actual search for a ray intersection with the particle data set. Figure 2.5 depicts the entry of a ray into a kd-tree and subsequent entries into the left and right subtree.

The traversal of the tree starts at the top root node and *descends* into child trees until a leaf-node (a subtree of size $1$) is encountered or no subtree is entered by the ray. The latter occurs when the ray does not intersect the subtree's enlarged bounding box or

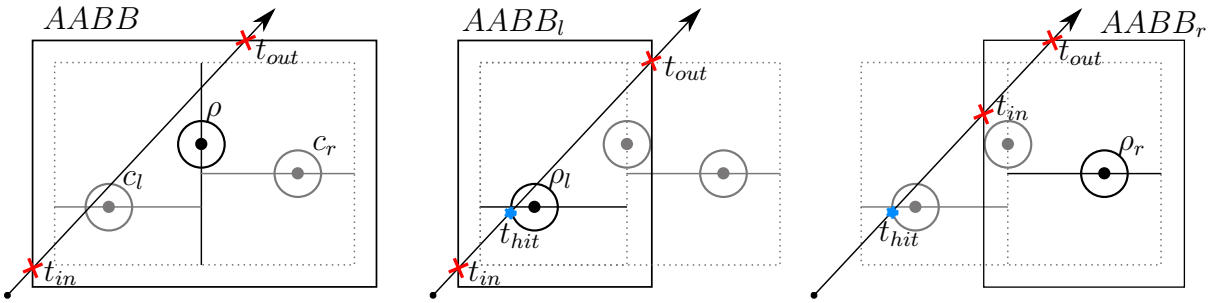



**Figure 2.5:** Ray traversal in the pkd-tree based on bounding box intersection. Left: The ray intersects the extended bounding box of the pkd-tree, and thus the particle node $\rho$ splitting the tree is tested for ray-intersection. Middle: The bounding box of the left child tree is derived using the parent node $\rho$. The ray enters the subtree and intersects the new root node $\rho_l$. Right: After traversal of the left subtree, the ray is tested for entry of the right subtree. Because the nearest sphere-intersection the ray encountered so far lies in front of the bounding box entry point, the right subtree need not be traversed.

when the ray's current closest hit lies before that intersection point. The traversal then *ascends* upwards until an intersected, but yet unvisited sibling-subtree is encountered, or until the kd-tree's top root node is reached again. So the ascend-traversal turns into a descend-traversal when a parent node is encountered of which both children intersect the ray, one child has been visited (where the ascension comes from), and the other child has not yet been visited. For the traversal to work correctly, the algorithm needs to maintain a stack which encodes the following information: bounding box of parent node, whether to visit the parent's other child tree after traversal of the current child, and whether the current subtree is the left or right child of its parent. When ascending, the parent node's information is pushed to the stack. When descending, that information is popped from the stack and used to restore the traversal state of the parent node. Therefore, the size of the stack is defined by the depth of the kd-tree, which is logarithmic in the number of particles. Due to the regular memory layout, the address of a child tree's root node (and therefore its position in the particle data array) can be computed directly by means of

$$child\_size = \frac{parent\_size - 1}{2}$$
$$child\_address_s = parent\_address + 1 + s \cdot child\_size$$

where $s \in \{0, 1\}$ decides whether the left or right child's address is computed. In a similar fashion, other information like bounding box and split axis of the next visited node can be directly derived from the state of the current node and the top entry on the stack.

Compared to the original paper, notable differences of the GPU implementation arise from the fact that OpenGL shaders do not allow recursion and arbitrary memory allocations. So the OpenGL fragment shader implementation needs to guarantee an upper bound of stack entries during traversal. Shaders should in general use few resources to allow faster processing. This is realized by always having only one set of tree traversal data (tree size, current subtree address, bounding box, split axis) and constantly updating it with newly computed information for child and parent nodes. For minimal memory footprint, the traversal stack is implemented using one $(float, bit, bit)$-tuple per stack entry, which leads to the stack consisting of one array of `floats` and two `unsigned int` variables serving as bit-arrays for the stack's bit values. The `float[]` array stores the bounding box coordinate which would otherwise get lost when shrinking the parent tree's bounding box to the bounding box of a child tree. The `unsigned int` variables contain information on the currently traversed tree side and whether the other side is queued for traversal. These variables are manipulated using GLSL *bitfield* functionality, which allows to set or read individual bits in integers. Using *bitfield* operations turned out to be faster than the alternative of using *integer arrays*. Because OpenGL does not guarantee 64-bit integer support, using bit operations on *unsigned integers* limits the stack to a maximum size of 32 entires. While this still allows to handle data sets of size up to $2^{32} \approx 4.29 \cdot 10^9$, for smaller data sets the stack size should be chosen as small as possible to improve performance by reducing memory consumption of fragment shader executions.

The tight packing of subtrees in memory allows to halt the kd-tree traversal at any time and switch to a linear traversal of the subtree's contents, i.e. the particle data contained in that subtree. This way, traversal has not to descend to subtrees with size $1$, but can instead halt traversal at subtrees of some *cutoff* size $c$. This effectively redefines leaf nodes from containing one particle to containing $c$ particles in the corresponding subtree. The particles in that subtree can then be tested for intersections with the ray in a brute-force manner, and the tree traversal algorithm can then continue by ascending the tree. This optimization improves the algorithm by replacing many alternating descending and ascending steps at leaf nodes with one linear sweep over the corresponding particle data, at the cost of testing particles for intersections which might not have been tested otherwise. Depending on the size of the data set, introducing such a *cutoff* at lower subtrees tends to improve performance. Another property of the chosen memory layout is that subtrees can be individually uploaded to GPU memory, which is important when handling data sets which are too big to fit into GPU memory.

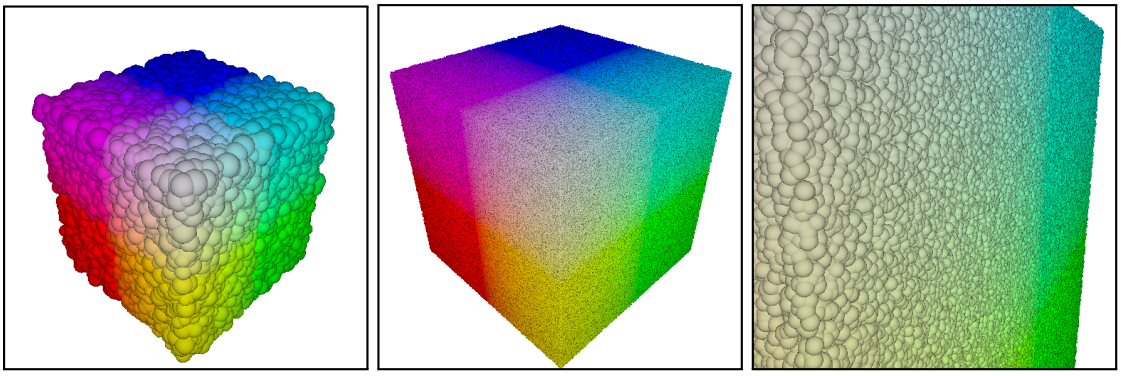


**Figure 2.6:** Particle data sets used for evaluating pkd-tree traversal performance on the TITAN and Radeon R9 GPUs. Left: 65 thousand randomly positioned particles in a $[-1, 1]^3$ cube with random radii in the range $0.01 - 0.1$. Middle: The same cubical setup, but with 10 million particles with radius $0.01$. Similar cubical data sets with sizes 1 million up to 134 million are used to stress-test the pkd-tree performance on the Radeon R9. Right: Closeup of the 10 million particles cube.

### 2.2.2 Performance Measurements and Implications

The evaluation of the implemented GPU pkd-tree traversal shows mixed results. Table 2.9 shows measured frame times in milliseconds for pkd-tree traversal on a GeForce GTX TITAN and an AMD Radeon R9 200, for a rendering resolution of $512^2$. Roughly $65$ thousand particles are ray casted using the pkd-tree, the resulting frame times amount to $0.3 - 1.6$ frames per second on the TITAN and $5.5 - 101$ frames per second on the Radeon R9. See Figure 2.6 for images of the tested data sets. Although the subtree-cutoff optimization improves performance for small cutoff sizes (cutoff tree size 64 up to 512), the linear memory traversal used by that optimization decreases performance for larger cutoff values. All in all, the pkd-tree traversal performs worse than rasterizing sphere glyphs for a data set of $65$ thousand particles. For larger data sets, frame times on the TITAN only become worse and benchmarking is not possible.

On the Radeon R9 GPU however, data sets consisting of up to $134$ million particles can be rendered with $4.4$fps using pkd-tree traversal. As shown in Table 2.10, frame times for sphere rasterization overtake frame times for pkd-tree traversal for data sets larger than $67$ million particles. At that particle data size, geometry processing for particles becomes more expensive than the evaluation of the pkd-tree in fragment shaders. For data sets smaller than $67$ million particles however, sphere glyph rasterization is much more efficient on the Radeon R9 GPU.

The reason why the Radeon R9 handles pkd-tree traversal much better than the TITAN probably is related to different optimizations of the GPU architectures. Fragment shaders are usually meant to execute short programs which make few memory accesses to predefined memory locations. In contrast to this, the pkd-tree traversal runs in a long `while` loop with many `if` clauses, needs increased memory resources for the traversal stack and makes a lot of calculations and arbitrary memory lookups. Furthermore, the increased branching of the tree traversal algorithm probably does not favor the *single instruction, multiple data* shader execution model of OpenGL. For some reason, the hardware of the Radeon R9 can better handle the random memory access pattern and long fragment shader executions of the tree traversal.

Despite its good performance on the Radeon R9, the pkd-tree also has some downsides. The most important downside is that the algorithm probably will not perform good an all OpenGL hardware, as the performance on the TITAN shows. As the $65$K data set measurements suggest, breaking down a large pkd-tree in lots of smaller subtrees will reduce the performance benefit of the pkd-tree, as direct rendering of smaller particle sets is faster than traversal of small pkd-trees. This leads to the problem that the pkd-tree held in GPU memory should be as large as possible. But breaking a large tree into smaller subtrees would be required to allow controlling usage of GPU memory and to use techniques like view-frustum culling to reduce memory- and data upload-demands. As the performance of traversing the contents of a pkd-tree depends on the rendering resolution, pkd-trees will not scale to larger resolutions as well as rasterization. However, the pkd-tree implementation may benefit from utilizing coherent ray-packet traversal [GIK+07; WIK+06] and other low-level optimizations to further improve performance. Because of the tree-size restriction used to design the tree traversal on the GPU, further techniques need to be incorporated into the algorithm to allow rendering of arbitrarily-sized data sets. The GPU pkd-tree traversal may further benefit when ported to GPGPU frameworks like CUDA and OpenCL, where arbitrary memory access patterns during tree traversal might be handled better.

On basis of the performance numbers presented for the TITAN GPU in Table 2.9 and because of the mentioned downsides of the pkd-tree, I consider straight forward sphere glyph rasterization the better approach to render spheres in OpenGL. But this does not mean that similar GPU data structures are out of the question for particle rendering. For example, grid- and octree-based particle ray casting on the GPU is reported by [LBH12] and [RCSW14] to have better performance than sphere glyph rasterization. [LBH12] render biological structures with up to 148 million atomic particles with $15$fps by traversing a grid on the GPU and testing particles contained in a grid cell for ray intersection. Using instancing of reoccurring biological structures, they achieve visualization of up to 10 billion atoms. However, instancing requires knowledge about structures contained in the data set, and this does not solve rendering of large data sets where no multiple instances of objects occur. Furthermore, it is not clear how much

memory the used grid structure requires in relation to the particle data size[6]. [FKE13] optimize the on-GPU grid traversal approach by [LBH12] and report rendering of up to $25$ billion atoms at $3.6$fps, but they too do not include numbers about additional memory consumption of their grid structures in GPU memory. [RCSW14] sample fluid particle data into a hierarchical voxel data structure where particle positions are mapped to occupied voxels on the lowest grid resolution level, encoded by single bits representing occurrence of particles in certain grid positions. By ray-traversing this voxel structure on the GPU they can render dynamic fluid simulation data sets of up to 500 million particles at interactive frame rates. Since the goal of this work is to visualize particle data sets without any reduction or sampling of the particle data, the hierarchical binary voxel representation by [RCSW14] is not considered a reasonable approach to achieve that goal. So other approaches of evaluating data structures for particle rendering on the GPU not really fit our needs and goals, and the pkd-tree turns out to be efficient only on very large data sets and on specific OpenGL-capable hardware.

In conclusion, the pkd-tree traversal on the GPU could be improved by further optimizing and extending it to support arbitrarily sized data sets. Due to the traversal-overhead in the fragment shader, pkd-trees need to have a certain size to be faster than sphere glyph rasterization on the GPU. If the particle data set does not fit into GPU memory, this leads to the problem of correctly and efficiently rendering smaller subtrees, while somehow maintaining the benefits of the pkd-tree data structure. Because of these downsides, straight forward sphere glyph rasterization is considered the better choice for rendering with OpenGL.

[6] [LBH12] report a $9.8$MB memory requirement for grid and particle data containing $900,000$ particles with $16$bit floating point positions. Assuming particle radii to also have $16$ bit representation, the amount of memory needed for particle positions and radii can be calculated to be $7.2$MB. This suggests a memory overhead of $26\%$ for the grid structure itself. For very large data sets, such a data structure would need too much overhead in GPU memory, especially since the grid size needs to increase with the particle data size to keep the amount of particles per grid cell low and thus achieve good performance.

| Milliseconds per Frame depending on Subtree Cutoff | | | | | | | |
|---|---|---|---|---|---|---|---|
| Subtree Cutoff | 1 | 64 | 128 | 256 | 512 | 1024 | 2048 |
| TITAN | 3045 | 685 | 655 | 752 | 903 | 1051 | 1341 |
| Radeon R9 | 179 | 24 | 16.6 | 13.1 | 9.9 | 10.9 | 16.8 |

**Table 2.9:** Pkd-tree rendering times in milliseconds per frame on a GeForce GTX TITAN and Radeon R9 200. The data set rendered consists of $65,535$ particles equally distributed in a $[-1,1]^3$ domain and with a random sphere radius in the range $0.01 - 0.1$. The rendering resolution was $512^2$. The measurements show that introducing a cutoff which switches to linear traversal of memory for subtrees of a certain size improves performance significantly. However, the pkd-tree approach can only render with $0.6-1.6$ frames per second on the TITAN, while the Radeon R9 achieves $5.5 - 101$ frames per second. For this very small data set and rendering resolution, both systems are slower than rasterizing sphere glyphs (TITAN: 625fps, RadeonR9: 526fps). The bad pkd-tree performance of the TITAN may be due to a hardware architecture that does not support many SSBO memory lookups and long running fragment shaders well.

| $k$ | 20 | 22 | 24 | 26 | 27 |
|---|---|---|---|---|---|
| # Particles $n$ | 1.048.575 | 4.194.303 | 16.777.215 | 67.108.863 | 134.217.727 |
| Data Size | 16MiB | 64MiB | 256MiB | 1024MiB | 2048MiB |
| Pkd-Ray Traversal | 73.6 | 87.4 | 152.4 | 190 | 225 |
| Sphere Rasterization | $3.9_{3.1}$ | $13.1_{12.1}$ | $47.8_{46.2}$ | $198_{195}$ | $656_{648}$ |

**Table 2.10:** Pkd-tree traversal and sphere rasterization times in milliseconds per frame for different particle data sizes. Particles with radius $0.01$ are equally distributed in a $[-1,1]^3$ cube, the rendering resolution is $1024^2$ and all particles are inside the view frustum. $k$ determines the number of particles in the tree according to $n := 2^k - 1$. While the rasterization approach is better for $k < 26$, the pkd-tree ray-traversal algorithm renders data sets with 67 million or more particles faster. For Sphere Rasterization, the geometry processing overhead when all particles are outside the view frustum is given by the numbers in subscripts. For very large data sets, traversing pkd-trees in fragment shaders becomes faster than processing the particle data in vertex and geometry shaders. The Radeon R9 did not have enough video memory to further increase the particle data size. The single-threaded pkd-tree construction for the 134 million particles data set took 8 minutes on an Intel Core i7-3820 @ 3.6GHz.

# 3 Organizing Large Data Sets

The goal of this work is to render large data sets interactively. When data sets become too large to entirely fit into GPU memory, an explicit and efficient CPU-side management of the data becomes necessary. Organizing large data sets can consist of a preprocessing to reduce the data size, e.g. by compressing the data or finding appropriate hierarchical level-of-detail representations for rendering [EBN13; FSW09; GK15; GM04; GNL+15; HE03; KBR+12; RTW13]. Besides handling the size of data, efficient rendering needs to rely on data structures which manage the complexity of the scene itself by spatially sorting objects and providing the rendering algorithms with the actually visible data. The spatial sorting should be able to identify objects which are visible to the viewer (i.e. provide frustum and/or occlusion culling) and it should be able to sort the visible primitives according to their importance for the currently rendered image.

For rendering particle data as spheres, the general assumption of this work is that the nature of the data is space-filling, opaque and not necessarily dense. The data is assumed to be space-filling in the sense that individual particles contribute to complex three-dimensional structures in the scene, where each particle may occlude several other particles, but no large group of particles can be trivially identified to form a coherent structure (e.g. a connected surface) that occludes other parts of the scene. This implies that the scene may have a high depth complexity with a lot of individual objects. Individually, the majority of particles may contribute little to the rendered image (occupying only few pixels per particle), but lots of particles will contribute to the appearance of structures in the image. This mixture of high depth complexity, lots of drawn objects, small individual image-contribution per object and non-trivial identification of formed structures in the scene leads to the situation that little to no guarantees about the data set can be made, and thus algorithms must be designed without assumptions about the data.

This chapter will investigate organizing the data set in a way such that the rendering algorithms can be efficiently provided with important data. The assumption made about the particle data is that the whole data set fits into main memory, but it not necessarily into GPU memory. The data set will not be reduced or sub-sampled in any way. The goal is to render the whole data set, but in a way such that important (i.e. directly visible) parts of the data set are rendered with higher priority. The particles are going

to be rendered as opaque spheres by the GPU, for which the CPU-side data structures developed in this chapter will provide a spatial sorting, depending on the current view of the scene. This CPU-preprocessing of the data set aims to use the parallelism of modern multi-core CPUs where possible. The *C++14* standard provides several features which allow for fairly easy cross-platform multi-threaded programming. So processing of particle data will be designed with the goal of using embarrassingly parallel algorithms where possible.

## 3.1 Using a Uniform Grid

As primary data structure I use a uniform grid which is overlaid over the particle data. The reasoning is that the structured nature of the grid helps to derive information about the presumably unstructured distribution of particles in the scene. The grid partitions the data set into disjoint regions which have a fixed spatial extent. Also, the construction of the grid, i.e. sorting the particle data into the grid, can be done efficiently in linear time complexity, which is important when handling large data sets. The grid cells in the grid data structure can be accessed in constant time without having to traverse some sort of hierarchical data structure, which makes it suitable for easy task-parallelism. Furthermore, a grid does not need much additional meta data which might increase memory consumption.

The grids cells, holding subsets of the particle data, can further be used to find abstractions of the contained data, e.g. compute compact representations for large clusters of particles. Furthermore, simple statistics like occupation of individual grid cells can be easily computed and used to aid rendering. Cell densities might also be used for a volume rendering of the data set, which is a natural use for data sorted into a grid [RCSW14; RTW13; ZD15]. In this work however, no volumetric approach for rendering is chosen. The grid is primarily used to organize particle data spatially and to derive a rendering order of scene contents based on grid properties, much in the style of [GRDE10] who partition particle data into a grid to implement efficient culling of occluded data set regions. For this reason, the term *voxel* is avoided in this work when referring to grid cells. Instead, grid cells will be referred to as *bricks*, because the grid cells literally contain smaller chunks of the whole data set. The particles contained in bricks will be handled and rendered as individual particles, i.e. opaque spheres. The grid, which holds bricks in its cells, is not assumed to be densely filled, which means that the grid may have cells which do not contain any particle data. The resolution of the grid is chosen to roughly partition the particle data domain into a few thousand or ten thousand bricks, so the goal is not to build a high-resolution grid. Depending on the spatial extent of the particle data set, the grid resolution along a dimension will be chosen coarsely, in the range of $30 - 175$ (e.g. a grid of $30^3$ resolution), rather than a resolution of $1000$ or upwards.

### 3.1.1 Grid Construction

Let $\{p_1, \ldots, p_n\} \in \mathbb{R}^3 \times \mathbb{R}_{>0}$ be the particle data set, where each particle consists of a position in space and a radius. A uniform grid is fitted into the axis aligned bounding box $B = (B_{min}, B_{max}) \in \mathbb{R}^3 \times \mathbb{R}^3$ of the particles' positions, where $B_{min}$ and $B_{max}$ are corners of the bounding box containing the minimum and maximum coordinates of

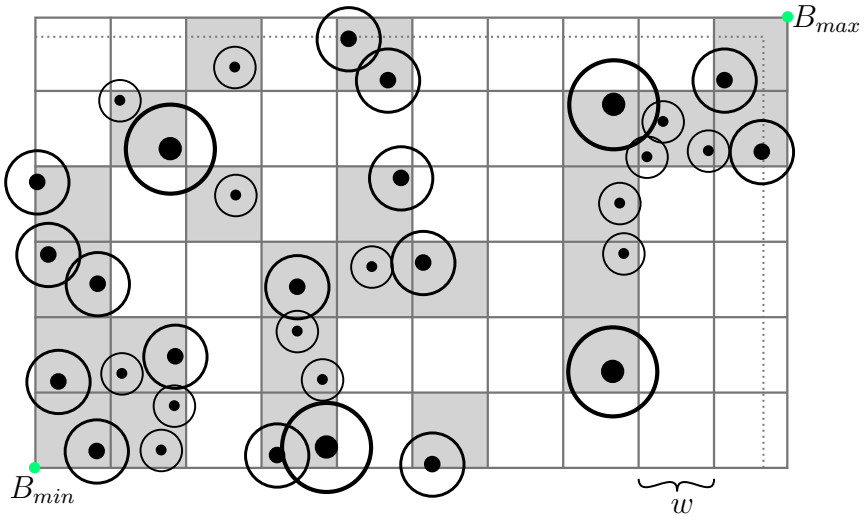



**Figure 3.1:** The uniform grid partitions the particle data into bricks. Depending on the data set, some grid cells may stay empty, in which case no corresponding bricks for those cells are considered.

the bounding box sides, respectively. The resolution of the grid is determined by a user-defined target resolution $d_{max}$, which is going to be the grid's resolution along the bounding boxes longest side $s_{max}$. From $s_{max}$ and the target resolution, the grid width $w = s_{max}/d_{max}$ is computed, which defines the size of the grid cells (referred to as *bricks*), and therefore defines the final three dimensional resolution of the grid inside the bounding box. This yields the grids final dimensions $d = (d_x, d_y, d_z)^\intercal \in \{0, 1, \cdots, d_{max}\}^3$. To guarantee numerical stability of further floating point operations, the grid width is adjusted to be slightly larger, in a way such that the condition

$$\begin{aligned} w' &> w \\ B'_{max} &= B_{min} + d \cdot w' >_{pointwise} B_{max} \end{aligned}$$

is fulfilled, i.e. the grid should contain the bounding box (and thus the particle data) conservatively, such that particle containment in the conservative bounding box in always fulfilled when computed in floating point arithmetic. The slightly larger grid width $w'$ and the conservatively enclosing maximum bounding box corner $B'_{max}$ are then used for all further calculations, replacing the old values of $w$ and $B_{max}$, respectively. A brick $b_{i,j,k}$ at position $(i, j, k)^\intercal \in \{0, 1, \cdots, d_{max} - 1\}^3$ in the grid is then defined by the bounding box induced by its minimum and maximum corner in the grid, $(b_{min}, b_{max}) = (B_{min} + (i, j, k)^\intercal \cdot w, B_{min} + (i + 1, j + 1, k + 1)^\intercal \cdot w)$.

Sorting the particle data into the grid is done by mapping the particles' 3d-positions into the grid bricks they are contained by. No new memory is allocated to hold the sorted data, but instead the data is sorted in-place using histogram-sort, a variant of

bucket-sort, which groups particles falling into the same brick into contiguous partitions in memory. Not allocating a second buffer in size of the particle data is important when dealing with data sets which already barely fit into main memory. The histogram-sort needs to loop over the particle data array twice, which improves computation time significantly compared to a recursive sorting algorithm like merge-sort or quick-sort.

In the first pass, the number of particles that fall into a brick is determined. Also, other metrics like brick density can be accumulated. Based on the first pass, the size and location of brick partitions in memory is derived. A second pass traverses the particles array front to back and swaps the currently visited particle into the memory position where particles of the corresponding brick are collected. Each particle gets swapped at most once into its brick's partition of the array, and after each swap one more particle is correctly sorted into its own brick. Each brick holds a partition-pointer into the particle array which indicates the end of the brick's partition in memory. Initially, all partition-pointers point at the first position of the brick's partition in memory, as established by the first pass determining brick partition sizes, which also dictates where each brick's partition starts in memory. After a particle has been correctly assigned to its brick (either by swapping or because it was already correctly sorted), the partition-pointer of the brick gets increased to the next position in memory, pointing at the yet unsorted particle after the brick's partition. Particles which need to be swapped into their corresponding brick-partition are swapped with that last, yet unsorted particle after the brick's partition. The algorithm terminates after all brick's partitions have grown to their final size. Since each particle gets swapped into its own brick partition at most once, and since each brick's partition pointer gets increased as often as the brick's partition has particles, the runtime of the algorithm is determined by the number of partition-pointer increases. This results in a runtime that takes

$$\sum_{b \in Grid} \mathcal{O}(|b|) = \mathcal{O}(\sum_{b \in Grid} |b|) = \mathcal{O}(|\{p_1, \ldots, p_n\}|) = \mathcal{O}(n)$$

steps, where $b \in Grid$ denote occupied bricks in the grid, $|b|$ denotes the partition-size of a brick, and $p_i$ denote the particles contained in the data set.

While the first pass can be parallelized with little synchronization overhead, a parallel implementation of the sorting is not obvious, and thus the second pass is bound by single-core performance and memory bandwidth. The mapping of a particle's position $(x, y, z)^\intercal \in \mathbb{R}^3$ into its brick is done by the equation

$$(i, j, k)^\intercal = ((x, y, z)^\intercal - B_{min})/w,$$

where the 3d-brick index $(i, j, k)^\intercal$ can be used to directly index a brick in some array of bricks by linearizing it. However, due to floating point inaccuracy the position of the particle is double-checked to actually be contained by bounding box of the calculated

brick, and if necessary is assigned to the adjacent brick it spatially falls into. This correction is done so other algorithms can rely on the particles inside a brick to be actually contained within the the brick's bounding box. Note that this sorting into the brick merely assigns the particles' positions into bricks, it does not consider the particles' spatial extent as spheres. So particles inside one brick may spatially overlap other bricks when drawn as spheres. Usually, the radius of particles is very small with respect to the overall scale of the scene. So this brick-overlap often will not be a problem for other algorithms. Furthermore, the brick-overlap can be restricted to only affect adjacent bricks by choosing the grid width $w > 2 \cdot r_{max}$, where $r_{max}$ is the maximum radius of all particles in the data set.

Depending on the data set, some bricks in the grid may not contain particles at all. This means that in general, the grid may be *sparsely occupied*, and only occupied bricks are relevant for further considerations. Interesting meta-data of a brick includes:

- Axis aligned bounding box $(b_{min}, b_{max})$ representing the brick.
- Number of contained particles.
- Memory range in which particles reside.
- Density of brick, i.e. volume of contained spheres in relation to brick volume.
- World position in the scene, represented by the brick's center

$$b_c = b_{min} + (b_{max} - b_{min})/2.$$

- Maximum radius $r_{max}$ of all particles inside the brick.

Because the sorting of the particle data into the grid is just an in-place reordering, no additional data structures which scale with the data set size are necessary. The grid structure itself maintains meta data information per brick as described above, but the memory consumption of this scales with the grid dimensions. For example, assuming that the mentioned meta data consumes $13 \cdot 4$ *bytes* per brick, the total memory overhead for meta data is roughly $46$*MiB* for a fully occupied grid with dimensions $100^3$. This memory requirement seems negligible compared to the size of the data sets which are going to be managed, which will be in the order of several hundred megabytes up to several gigabytes (or eventually using all available system memory). Meta data about the grid, like grid dimensions and memory locations of brick contents, can be stored in a separate file independently from the particle data, or in dedicated meta data fields of a suitable particle data file format.

The grid is not intended to have a high resolution, it should only divide the data set into not too big, reasonably sized chunks of particles. This is in contrast to volume data grids which capture high-frequency details of data sets by choosing higher resolutions.

Depending on the particles' data domain, a grid resolution in the range of $30 - 175$ along the longest domain axis gave reasonable results for partitioning tested particle data sets.

### 3.1.2 Frustum Culling

Frustum culling is a feature that should be available when visualizing large data sets. Although the data set may be fully visible any time, exploration of details in the scene will lead to large parts of the data set being outside of the view frustum. Using the grid data structure, frustum culling is realized on the level of bricks, instead of culling individual particles.

Computing frustum intersection and culling for axis aligned bounding boxes has been studied in detail [AM00]. So testing brick bounding boxes against the frustum directly is one feasible approach. However, correctly and efficiently implementing the geometrical intersections needed for those algorithms can be complicated, so I opted for a more robust and conservative frustum culling approach. So to simplify testing of frustum intersection and containment for bricks, I use the bounding sphere of a brick for frustum culling, defined by the brick's center $b_c$ and half-diagonal $b_{diag} = b_{max} - b_c$. To avoid explicitly constructing and testing the planes defining the view frustum, I compute frustum culling in clip space and in view space, i.e. after transforming the bricks' centers with the view- and projection-matrix for the current camera pose and viewport, respectively. These decisions for the frustum culling algorithm have no deeper meaning for the other algorithms used for data management or rendering, other than that this culling variant was fast to implement reliably. In the implementation code, frustum culling may be swapped for any other correctly functioning culling code that culls the brick-boxes geometrically correct. This will slightly reduce the number of bricks (and thus particles) processed by further steps of the rendering process.

Given a point $v^{view} = (v_x, v_y, v_z, v_w) \in \mathbb{R}^4, v_w = 1$ in homogeneous camera coordinates, the projection-transformation of that point into clip space in OpenGL is defined by

$$v^{clip} = \begin{pmatrix} \alpha \cdot v_x \\ \beta \cdot v_y \\ \gamma \cdot v_z + \gamma' \\ -v_z \end{pmatrix} = \begin{pmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & \gamma' \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix}$$

$$= \underbrace{\begin{pmatrix} 1/(ar \cdot \tan\frac{fov_y}{2}) & 0 & 0 & 0 \\ 0 & 1/(\tan\frac{fov_y}{2}) & 0 & 0 \\ 0 & 0 & -\frac{z_{near}+z_{far}}{z_{near}-z_{far}} & \frac{2 \cdot z_{near} \cdot z_{far}}{z_{near}-z_{far}} \\ 0 & 0 & -1 & 0 \end{pmatrix}}_{\text{projection}} \cdot \underbrace{\begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix}}_{v^{view}}$$

where $ar = \frac{width}{height}$ is the aspect ratio, $fov_y$ is the field of view angle in $y$-direction, and $z_{near}, z_{far}$ are the depth values of the near and far plane, respectively [Ahn13; Ric16]. The key observation is that the projection matrix applies a simple scaling to the input vector's $x$- and $y$-components, and that the vector's depth value in camera space carries over to the $w$-component in clip space. With this information, it is simple to check frustum intersections of spheres in clip space by checking whether the sphere's center is inside or close to the the clipping volume. Because the $z$-component in clip space behaves non-linearly with respect to the original view space value, only $xy$-planes at a certain $z$-value in the clipping volume can be used for considerations. Those $xy$-planes allow to reason about geometrical relations between spheres and view frustum like in view space, though. Testing a sphere against the near and far plane needs to be done using the $w$-component, which represents the sphere's distance to the camera in view space. Note that for vertices in front of the camera, $v_w^{clip} = -v_z^{view} > 0$ holds, since $v_z^{view} < 0$ due to the camera looking in negative $z$-direction.

In clip space, transformed bounding spheres defined by center $c = b_c$ and radius $r = \mid b_{diag} \mid$ are tested against the frustum as follows. Given a vector $v \in \mathbb{R}^4$, let $v_k$ denote the $k$-subvector of $v$, e.g. $v_{xy}$ is the subvector containing $(x, y)^\intercal$. Let $\leq_.$ be the pointwise comparison of vectors. Determine frustum intersection and containment by testing

1. If
$$-c_{www} \leq_. c_{xyz} \leq_. c_{www} \tag{3.1}$$
holds, the sphere is mostly inside the frustum. This is the clipping condition to clip vertices in clip space after vertex processing in OpenGL [Mar17, p. 435]. Return *inside*.

2. Let $\delta$ be a correction vector $\delta = (\delta_x, \delta_y)^\intercal$ for positive correction factors $\delta_x, \delta_y > 0$. If
$$-c_{ww} - \delta \leq_. c_{xy} \leq_. c_{ww} + \delta \text{ and } -c_w \leq c_z \leq c_w \tag{3.2}$$

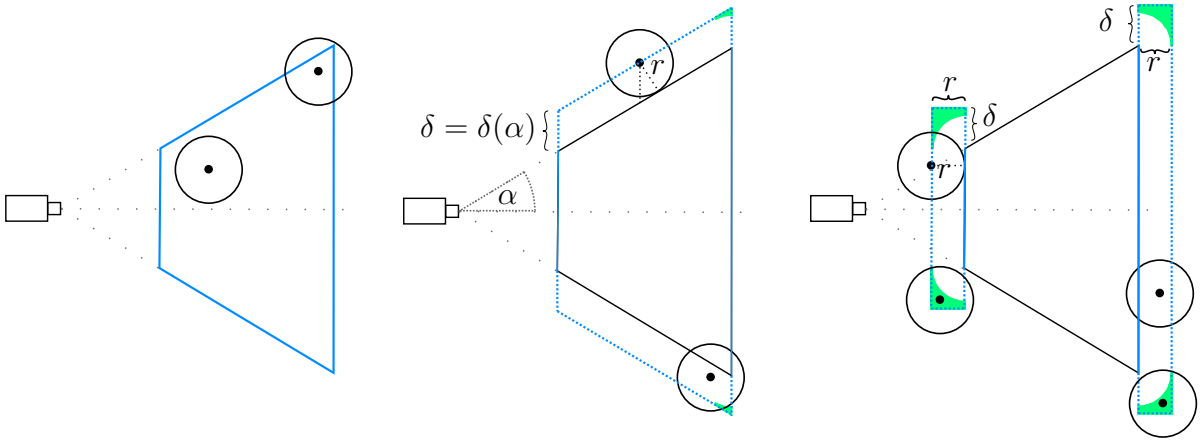



**Figure 3.2:** Testing sphere intersection against the view frustum. Left: Using the vertex clipping condition, the sphere's center is determined to be inside the view frustum in clip coordinates. Middle: If the sphere's center is outside the view frustum, the frustum is enlarged by an offset $\delta$ in $x$ and $y$ direction, which allows to again use the clipping condition to verify sphere-frustum intersection in clip coordinates. Right: Sphere intersection with the near and far plane needs to be handled separately. Again, the view frustum is enlarged by appropriate offsets (the sphere's radius $r$ and $\delta$ as before) to conservatively test the sphere. Note that at the corners of the enlarged area, some spheres will be falsely classified intersecting the frustum (green areas). Correctly handling these cases would require more complicated test conditions.

holds, the sphere's center is outside the view frustum, but the sphere intersects the view frustum at one of the planes *up, down, left, right*. Thus, return *inside*.

3. Let $z_{near}$ and $z_{far}$ be the distances of the *near plane* and the *far plane* to the camera, respectively, and use the correction vector $\delta$ like before. If

$$c_w + r \geq z_{near} > c_w \text{ and } -(z_{near}, z_{near})^\intercal - \delta \leq_. c_{xy} \leq_. (z_{near}, z_{near})^\intercal + \delta \quad (3.3)$$

or

$$c_w - r \leq z_{far} < c_w \text{ and } -(z_{far}, z_{far})^\intercal - \delta \leq_. c_{xy} \leq_. (z_{far}, z_{far})^\intercal + \delta \quad (3.4)$$

holds, the sphere is outside of the view frustum, but the sphere intersects the view frustum at either the near or the far plane. Thus, return *inside*.

The different conditions cover all cases in which a sphere is either inside the view frustum or intersects it. Figure 3.2 depicts the different tested conditions and the underlying geometrical properties. While Equation (3.1) tests whether the sphere's center is inside

the view frustum, the other conditions cover the cases where the sphere's center is outside the frustum, but the sphere itself intersects the frustum. Equation (3.2) checks whether the sphere intersects the frustums *top*, *down*, *left* or *right* planes by shifting the sphere's center by a correction value $\delta$ in $x$ and $y$ direction. The values $\delta_x, \delta_y$ are the minimum distance the center needs to be shifted in $x$- and $y$-direction, respectively, such that the sphere's silhouette touches the frustum [Dan02]. The factors $\delta_x$ and $\delta_y$ depend on the sphere's radius and the field of view angles in $x$ and $y$ direction, respectively. As according to [Unk11], the components of $\delta$ can be computed in view space by

$$\delta_k = \frac{r}{\cos \alpha_k}.$$

For $k \in \{x, y\}$, the angles $\alpha_x$ and $\alpha_y$ represent the half angles of the $x-$ and $y-$field of view, respectively. $\delta$ can then be transformed to clip space using the projection matrix, correctness of $\delta$ carries over to clip space because $x-$ and $y-$dimensions are only scaled by the projection from view space to clip space. Since the $z$-component behaves strange in clip space, it is easier to use the $w$-component $c_w$ to check for intersection with the near and far plane. This is done in Equation (3.3) and Equation (3.4) by testing whether the center's distance to one of the planes is smaller than the sphere's radius. To only accept spheres which are directly in front or behind the frustum, a similar verification as in Equation (3.2) is used to discard sphere's which are too far away in $x$- or $y$-direction. Note that in the third case, spheres will be accepted as *inside the view frustum* although they do not actually intersect it. This happens due to a simplification of the intersection-condition at the edges of the near and far planes: to be correct, a spherical shape induced by the spheres radius should be used to test for sphere intersection, but instead rectangular boxes are used. See Figure 3.2 for an illustration of the introduced error (areas marked in green).

To account for particles which might spatially overhang a brick's bounding box, the sphere's radius used for culling can be extended by the maximum particle radius in the brick. Since each brick is handled individually, the frustum culling can be parallelized easily. As output, the culling algorithm creates a list of visible bricks which then can be further processed and rendered. For example, the already transformed brick centers can be used to sort the bricks by depth, since a point $c$ in clip space holds the distance to the camera in the $w$-component. Because a front-to-back sorting of scene elements is desired and will be central in further parts of this work, computing the camera distance to bricks is necessary anyway. This computation is equivalent to applying the camera matrix to the brick's world position. Thus, transforming brick positions to clip space provides the depth in camera space for front-to-back sorting and a frustum culling of bricks, at the cost of one matrix multiplication with the combined view-projection matrix.

## 3.2 Visibility and the Occlusion Graph

In this chapter, a visibility sorting for bricks is derived from their occlusion relations in the grid. This way, bricks can be rendered according to their importance and contribution to the current image.

Computing visibility is one of the key problems in computer graphics, and has therefore been studied extensively [BW03; GKM93; TS91]. The visibility problem is to determine which surfaces (or parts of surfaces) in the scene are visible from a given viewer position. Solving visibility in object space usually involves shooting rays into the scene and determining which surface they intersect first. To achieve better performance of such algorithms, spatial data structures are used to accelerate the search for possibly intersected surfaces Modern rasterization hardware uses the z-buffer algorithm to solve the visibility problem [Cat74]. Z-buffering compares the depth values of fragments falling into the same framebuffer pixel, and only keeps the fragment with the smallest depth value, which is thus visible to the viewer and occludes all fragments behind it.

The goal of *occlusion culling* is to determine and discard non-visible objects and surfaces from rendering because they are occluded by other objects in the scene, and thus not visible to the viewer. While the z-buffer operates on a per-pixel basis to guarantee a correct image, occlusion culling operates on whole objects and aims at reducing overall rendering cost while maintaining correctness of the image. Modern GPUs can be used to assist occlusion culling using *occlusion queries* . An occlusion query renders geometry on the GPU, tests the resulting fragments against the z-buffer and reports whether any fragments would be written to the framebuffer, i.e. whether the drawn geometry would contribute to the image. Based on the result of the occlusion query, the renderer can decide whether to actually draw the geometry. However, this occlusion feedback from the GPU to the rendering application running on the CPU introduces latency, since final rendering of geometry is postponed until occlusion query results are available. The impact of this latency on rendering performance can be reduced by exploiting spatial and temporal coherence of drawn objects [BWPP04; GRDE10]. However, for a simplified rendering algorithm and renderer design it is desirable to handle occlusion culling mostly or exclusively on the CPU, supplying the GPU only with geometry that is going to be rendered.

When rendering large amounts of particles, computing occlusion culling on the level of individual particles is not practical due to the size of the data set. Besides, computing the set of visible particles which occlude others would solve the original task of rendering the particle set. Nevertheless, as a method of scene management and data reduction, some kind of approximate culling and sorting of the particles in the scene is required. The idea is that parts of the scene which highly contribute to the final image should be

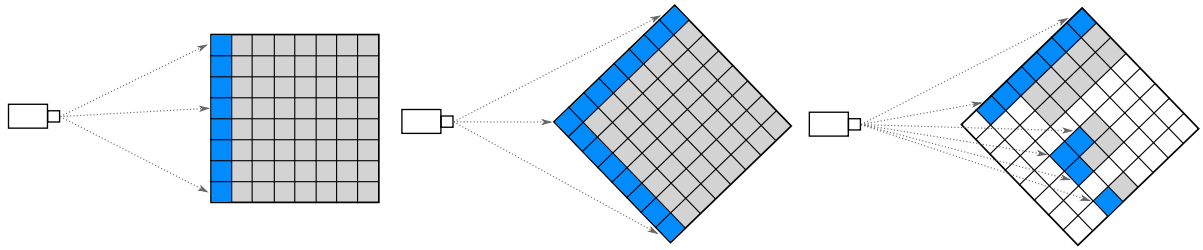


**Figure 3.3:** Illustrating the importance of visibility-based object sorting. Occupied bricks of the grid are marked in gray and blue. It is desirable to render directly visible bricks (blue) first. Rendering bricks based on their distance to the camera does not respect their visibility and occlusion relations. Left: For some scene configurations, rendering the bricks front-to-back draws them in the correct order, as directly visible bricks are rendered first. Middle: To render directly visible bricks (blue) first, simple depth-sorting does not always suffice. When sorting only by depth, a large part of the data set is going to be drawn (gray bricks) before the most right directly visible bricks. Right: When bricks are sparsely scattered in the grid, it is desirable to determine and draw directly visible bricks (blue) before all others (gray bricks).

rendered first, and less important parts of the data set can be drawn with lower priority, or discarded from rendering entirely.

The sorting of the scene will be done on the level of individual bricks of the grid, which have been determined as potentially visible by the view frustum culling algorithm. Note that although the bricks are cells of a uniform grid, not every cell of the grid must contain a brick with particles, i.e. the bricks containing particles can be sparsely distributed in the grid. The goal of this chapter is to provide not only a front-to-back sorting of the bricks, but to assign the bricks a rendering order which is based on their visibility in the scene, and therefore draw the bricks according to their contribution to the final image. A simple front-to-back ordering not always satisfies that requirement, as is illustrated in Figure 3.3.

Because the distribution of particles inside a brick can be arbitrary and especially non-dense, it does not suffice to determine the set of bricks directly visible to the viewer. All bricks located further behind the first line of visible bricks may also contribute to the image, which is why employing occlusion culling on a per-brick-level would lead to an incorrect rendering of the scene. This implies a layered ordering of all bricks in the view frustum, with the first layer being the set of *directly visible bricks*, and the successive layers being defined as the set of bricks which are *behind the previous layer*. So basically, the bricks of layer $k$ *occlude* bricks of layer $k+1$, with the initial layer $1$ not being occluded by anything. In this context, the term *occlusion* can have one of two

meanings. Bricks can be thought of as just the set of particles they contain, in which case it is assumed that it is not feasible to compute the occlusion-impact the particles will have when drawn (i.e. deriving what other particles they will occlude). But when defining a brick containing particles as *occupied* (i.e. *an opaque box in space*, regardless of the actual amount of particles inside), the occupied bricks in the scene can be sorted by the degree of how many opaque bricks occlude them. This spatial sorting of scene objects, as implied by their occlusion relations between each other, can be expressed in terms of a graph, called *occlusion graph*.

The occlusion graph concept is used by [RK95] to model occlusion relations for identifying hand gestures. [GHLM05; GLM04] utilize the occlusion graph to compute correct transparency ordering of arbitrary objects on the GPU using occlusion queries. However, they do not actually compute an occlusion graph for scenes, but they use the graph concept to derive properties of a visibility sorting. For the purpose of correctly rendering volume data, several visibility ordering methods have been proposed which optimize for different kinds of meshes and topologies [CICS05; CMSW04; SMW98; WE97; Wil92]. A regular grid is used by [BS02] to compute conservative occlusion culling for objects, which is done by sampling objects into the grid and using fast grid traversal to compute occlusion between grid cells.

In this work, I am going to compute occlusion graphs on the CPU, based on the structure of the uniform grid. The occlusion graph will be used to determine a visibility sorting of the bricks in the view frustum. Given a set of scene objects $\{o_1, \ldots, o_n\}$ and a camera position $c$, the occlusion graph $G = (V, E)$ is defined as follows:

- The nodes $V$ of the graph are the scene objects and the camera: $V = \{o_1, \ldots, o_n, c\}$.
- The edges $E$ of the graph are defined by:
  $E = \{e = (i, j) \mid o_j$ is occluded by $o_i$ when looked at from camera position $c\}$.

The definition of the graph edges can vary, depending on the exact definition of occlusion. Objects can fully or partially occlude others, and one might differentiate between direct and indirect occlusion. By direct occlusion I mean that one object $o_i$ occludes another object $o_j$ iff: considering all other objects in the scene, when a ray is sent from some point on object $o_j$'s surface, the first object this ray will intersect is object $o_i$. Indirect occlusion modifies this condition in so far as that *all* objects which the ray intersects on its way to the camera occlude the initial object. So the occlusion graph for indirect occlusion results from the transitive closure of the graph resulting from direct occlusion. An object $o_i$ fully occludes another object $o_j$ when *all* rays starting from $o_j$'s surface intersect $o_i$ (i.e. one object is in the shadow of the other, if the camera position is interpreted as a light point source). Partial occlusion means that only *some* rays starting from one object intersect the other.

Although computing visibility (and therefore occlusion) exactly is computationally demanding in general, an occlusion graph for bricks inside a grid can be computed efficiently, as will be shown in the following. Building and evaluating the occlusion graph will be done entirely on the CPU for each new camera position after user interaction, in a multi-threaded fashion where possible. The occlusion graph can then be used to derive a rendering order (*visibility sorting*) for the bricks in the scene.

### 3.2.1 Computing Occlusion in the Grid

Given a set of occupied bricks from a grid and a camera position, in the following I will analyze how to compute visibility, or occlusion, in a uniform grid, for the purpose of constructing an occlusion graph. It suffices to consider how one side of a brick, i.e. a quad, is occluded by other occupied bricks in the grid. Occupied bricks are bricks which contain particles, and for the purpose of this consideration occupied bricks are assumed to be opaque cells of the grid. Occlusion between bricks could be computed by shooting rays from the camera into the scene (e.g. one ray per pixel in the image), traversing the grid, gathering the encountered occupied bricks per ray, and assigning the occlusion relations between bricks on that basis. However, this would be computationally demanding due to a large number of rays that scales with the rendering resolution. Also, rays with similar direction would essentially compute the same occlusion relations between bricks. A more suitable approach is to reverse the ray direction: starting from individual bricks inside the grid, shoot rays towards the camera and determine occlusion on basis of encountered occupied bricks during grid traversal.

Given a brick for which occlusion by other bricks is to be computed, at least one and at most three sides of the brick are oriented towards the camera. In the following, consider one of those sides $s$, and construct four rays from the corners of $s$ towards the camera. The four rays form a quadrilateral frustum which gets narrower towards the camera, as depicted in Figure 3.4. If the side $s$ is fully or partially occluded by another brick, at least one of the four corner rays must intersect that occluder, i.e. it will enter the brick's grid cell when traversing the grid towards the camera. Because all occupied bricks in the grid have the same dimensions, and because the frustum formed by the rays always is narrower than a bricks side, no brick can be fully enclosed by the frustum without at least one ray passing through the brick. Iff a ray intersects an occupied brick, this brick casts a shadow on the side $s$, as depicted in Figure 3.4. The shape of the cast shadow can not be arbitrary, because the brick sides projected onto $s$ are oriented according to the three coordinate axes, and thus the slopes of the shadow's boundary edges are determined by this projection. Furthermore, when multiple rays of the frustum enter the same brick, the shadows induced by the ray intersections merge. This property could simplify exact computation of partial occlusion between bricks, by simplifying

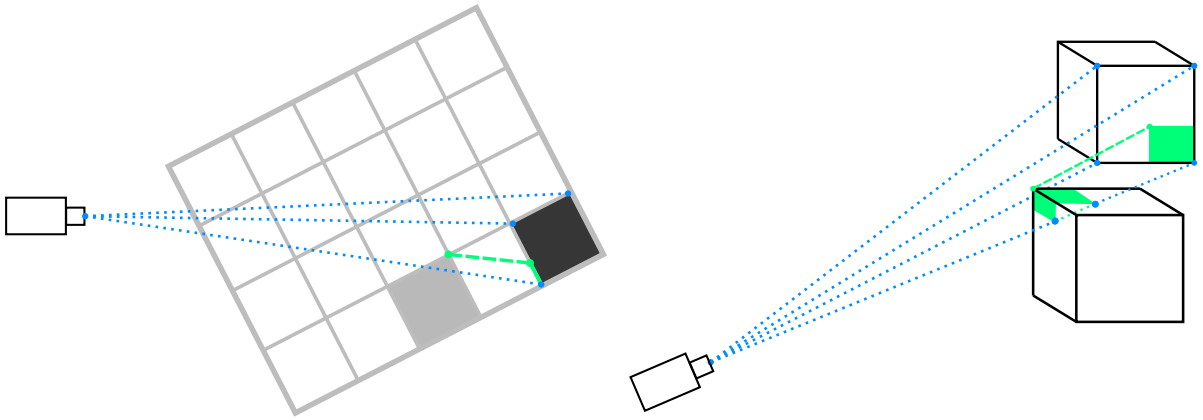


**Figure 3.4:** A brick-side is occluded if other bricks intersect the frustum spanned by the side's corners and the camera. Bricks in the frustum cast a shadow on the occluded side. Left: The black brick is occluded by the gray brick. The occlusion is determined by the gray brick's intersection of the blue frustum spanned by rays towards the camera. The occluding brick casts a shadow on the occludee (green line). Right: Three-dimensional case of occlusion by frustum intersection. The part of the occluder which is inside the blue frustum casts a shadow (in green).

bookkeeping of projected shadow shapes. There are three cases in which occlusion occurs. When one or two corner rays enter a brick, partial occlusion occurs. When three corner rays enter the brick, the fourth will also enter the brick, and thus full occlusion occurs. The inevitable entry of the fourth ray is due to the fixed relative orientation between frustum and bricks along the three coordinate axes, i.e. the fourth ray must enter a side of the brick because the frustum is not tilted.

The ray traversal inside the grid can be efficiently implemented using the DDA algorithm, which uses the ray's slope to determine the distance it can advance until it hits the next grid cell [AW87]. It turns out that although the corner-induced frustum helps to reason about the grid traversal and occlusion, using the same traversal strategy in an implementation leads to problems. The main problem is that no initial starting cell can be assigned to the four rays spawning at the corners, but each ray needs an initial cell from which the DDA grid traversal algorithm starts. For this reason, the exact traversal of the grid using the frustum approach needs to be simplified in so far as that rays start from some point on the side $s$ that is neither a corner nor on an edge. This way, the grid cell from which the ray unambiguously starts the traversal is the neighbor cell adjacent to the brick's side $s$. By shooting only one ray from the brick-side's center, the occlusion computation is simplified to determine only partial occlusion of at least $25\%$ per side.

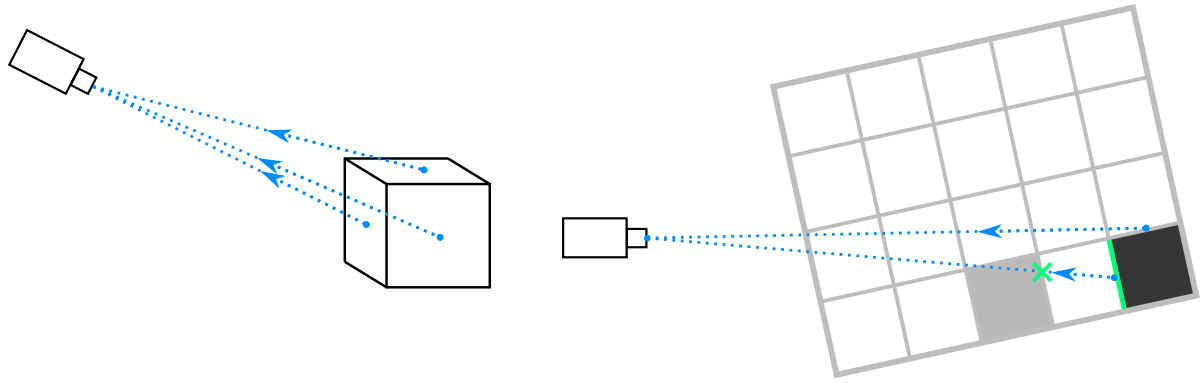


**Figure 3.5:** Computing occlusion between bricks is done by traversing the grid towards the camera with rays originating from the brick-side's centers. Left: As approximation to a correct occlusion computation, only one ray per visible brick side is shot towards the camera. Right: 2d-slice of grid traversal starting from the black brick. While the ray for the upper brick-side does not encounter occupied bricks, the lower ray enters one (gray brick). Encountering an occupied brick marks the initial brick-side as occluded by that brick (hinted by the green line).

For each brick-side oriented towards the camera, one ray is shot from the sides center to the camera, as illustrated in Figure 3.5. A ray traverses the grid until it enters the first grid cell that is marked as occupied. This occluding brick can either be the direct neighbor-brick from which the traversal started, or some other brick encountered after empty grid cells have been traversed. The occlusion relations determined during grid traversal of rays are saved into a graph structure, the occlusion graph, which consists of bricks as graph nodes $V = \{b_1, \cdots, b_n, c\}$ and edges $E = \{(b_i, b_j) \mid b_i \text{ occludes } b_j\}$ encoding the occlusion between bricks. For each brick $b_i$ determined as occluder of a side of brick $b_j$, one occlusion edge $(b_i, b_j)$ is produced. If no occluder brick is encountered by the ray, the brick-side is assumed to be directly visible to the camera, and a special edge $(c, b_j)$ is produced, with $c$ being the camera node and $b_j$ being the non-occluded brick. By limiting the number of rays emitted per brick to a maximum of three, at most three occlusion edges per brick will be produced, and thus memory for only $3n$ edges needs to be allocated, where $n$ is the number of bricks inside the view frustum. This upper limit on the amount of memory is important for an efficient implementation of the graph structure, because this way no costly memory-reallocations need to occur when adding new edges to the graph. Note that while some bricks may occlude many others, each brick individually is occluded by at most three other bricks.

The computation of the occlusion relations can be parallelized, as each brick is handled independently of all others. Synchronization between multiple threads computing occlusion needs to occur only when occlusion edges are added to the graph data

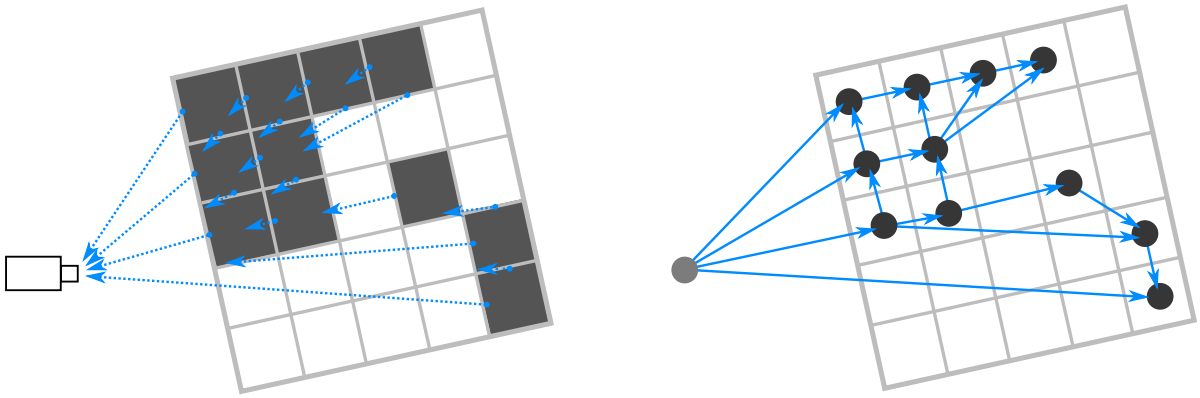


**Figure 3.6:** Occlusion graph resulting from a ray traversal of a two-dimensional grid. Left: Occupied bricks (in dark gray) in the grid shoot rays (blue) towards the camera, one ray per brick side. When the ray encounters an occupied brick, an occlusion edge for that brick is produced. Right: Occlusion edges (blue) between brick nodes (black) and camera node (gray) in the resulting abstract occlusion graph. The original grid is overlayed for better comprehension. Note that in densely filled regions of the grid, ray traversal immediately terminates at the neighbor brick.

structure, but this synchronization can also be postponed until all threads have finished their separate computations.

### 3.2.2 Deriving a Sorting of the Scene

Given an occlusion graph $G = (V, E)$ as constructed above, the goal is to derive a rendering order, or visibility sorting, of the bricks using the occlusion relations encoded by the graph. Because the bricks are cells of a uniform grid, no cyclic occlusion relations can occur. Thus, the graph $G$ is a directed acyclic graph with an unambiguous root node, namely the special camera node $c$, from which any brick node $b_j$ can be reached by traversing a sequence of occlusion edges. Each brick node has three incoming edges, and the only node without incoming edges is the camera node. See Figure 3.6 for a two dimensional example of an occlusion graph resulting from a grid traversal. Nodes which have no outgoing edges are leaf nodes. For a correct back-to-front rendering respecting occlusion relations of the bricks, one can iteratively draw the leaf nodes and then remove them from the graph, until only the camera node is left in the graph.

A front-to-back sorting respecting occlusion relations can be derived with a breadth-first search in the graph which assigns the brick nodes their minimal distance to the camera node. This partitions the graph nodes into levels, where a node at level $l$ has minimum

distance $l$ to the camera node. Thus, the camera node has level $0$ and directly visible bricks have level $1$. Given a brick $b_i$ at level $lvl(b_i)$, it propagates the shortest path to the camera node along its outgoing occlusion edges $(b_i, b_j)$ via

$$lvl(b_j) = \min_i \{lvl(b_i) + 1\},$$

where the brick $b_j$ is assigned the minimum distance from all incoming edges. See Algorithm 3.1 for pseudo code of level-assignment to brick nodes. Note that bricks in the same level may still partially occlude each other. But with the level sorting, this partial occlusion is given less priority than the occlusion relation to bricks nearer to the camera. If necessary, the occlusion relations between bricks of the same level can be respected by applying the same occlusion-sorting strategy to the set of brick nodes of one level.

**Algorithm 3.1** Level assignment to nodes in the occlusion graph.

```
1:  procedure BREADT-FIRST-ASSIGN-LEVELS(Occlusion graph G = (V, E))
2:      initially, set lvl(b) = ∞ for all nodes b ∈ V
3:      node c ← V.getCameraNode()
4:      lvl(c) ← 0
5:      node_queue.push_back(c)
6:      while node_queue.isNotEmpty() do
7:          node b ← node_queue.pop_front()
8:          for all nodes b' ∈ V reachable by an edge (b, b') ∈ E do
9:              if lvl(b') > lvl(b) + 1 then
10:                 lvl(b') ← lvl(b) + 1      // For each node, the level changes only once.
11:                 node_queue.push_back(b')
12:             end if
13:         end for
14:     end while
15: end procedure
```

The breadth-first traversal of the graph needs to maintain a queue of nodes and the level each visited node currently has. A parallel graph traversal would have to handle one or few nodes per thread, and the work done by a thread would consist of appending the next reachable nodes to the traversal queue. This would require a lot of synchronization between threads, where each thread does little actual work.

After nodes have been assigned a level in the graph, the brick nodes are sorted in two phases. First, the nodes are partitioned according to their respective node level. Secondly, each level internally is sorted according to the depth of the bricks in view coordinates. The brick's depth is computed during the view frustum culling stage, and is thus available without further effort. Other possible metrics to sort brick nodes of a

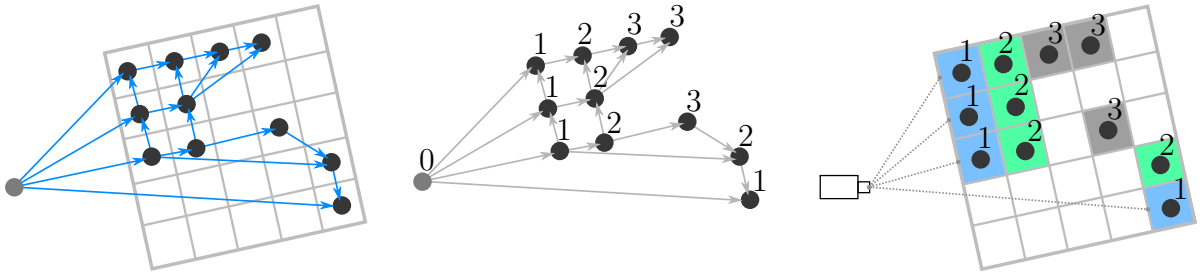



**Figure 3.7:** Occlusion graph and resulting node levels according to distance from the camera node. Left: The occlusion graph with overlayed grid. Middle: Assigned node levels according to shortest distance to camera node, as computed by the breadth-first traversal started at the camera. Right: Resulting brick rendering order according to node levels. Bricks with level $1$ (in blue) are directly visible and will be rendered first. The next batches of bricks are bricks with level $2$ (in green), and after that bricks with level $3$ (in dark gray). Note that bricks of lower levels may have greater spatial distance to the camera as bricks with higher level (bricks of level $3$, $2$ and $1$ on the right). But since the rendering order is based on visibility, this is desired.

level might include: the memory order in which the brick's data can be traversed fastest, screen footprint of bricks, occlusion between bricks of a level. Since all bricks have the same size, screen footprint is larger for bricks nearer to the camera, so screen footprint coincides with brick distance to the camera. The memory order of bricks has no meaning for the brick's contribution to the image, and although it might lead to faster rendering due to better cache behavior, the brick's spatial ordering due to the current camera position likely will not coincide with the memory order of the bricks. Sorting nodes of a level by occlusion relations among themselves would require further traversals of the occlusion graph limited to that set of nodes, which would require further processing time for computing the correct subgraph and traversing it. Thus, sorting nodes of a graph level by depth is the option with least processing effort which at the same time approximates the brick's contribution to the image best. Figure 3.7 depicts a rendering order of bricks resulting from node levels of the occlusion graph.

Another metric to derive a sorting of the bricks from the occlusion graph uses the brick's density, i.e. amount of contained particles. This can be seen as a generalization of the node-level approach described above, where node levels are not represented by discrete numbers anymore, but instead a brick's level is represented by the transitively accumulated real-valued density of all bricks which occlude it. Let the density of a brick $b$ be defined as the volume of all contained particle's spheres in relation to the brick's volume,

$$density(b) = \frac{\sum_{particle \in b} volume(particle)}{volume(b)}.$$

Densities are assigned to bricks using the same breadth-first traversal algorithm as before, but instead of computing shortest distance to the camera, densities are accumulated along shortest-paths from the camera to brick nodes $b$. The accumulated density for a brick node $b$ on the traversal-path is represented by the value $accum(b)$, the accumulated density-occlusion. Directly visible bricks are weighted with an initial density-occlusion value of $accum(b) = 0$, and a brick $b_j$ occluded by other bricks $b_i$ is assigned the minimal accumulated density of all occluding bricks as defined by

$$accum(b_j) = \min_i \{accum(b_i) + density(b_i)\}.$$

This density-sorting leads to an approximate front-to-back sorting of bricks, since bricks which are occluded by others will have larger accumulated densities than their occluders.

Consider Figure 3.8 for an example of density-based sorting of bricks. The accumulated density $0.7$ of the top right gray brick in Figure 3.8 is computed as follows: it is occluded by a brick of density $0.4$, which in turn is occluded by a directly visible brick of density $0.3$. The sum of those densities along the shortest path to the top right brick is $0.7$. The benefit of this sorting is that bricks which contain few particles and are occluded by bricks with small density get drawn with higher priority. However, the front-to-back order is relaxed in so far as that less dense bricks will be prioritized before dense ones, even if the latter are nearer to the camera or occluded by less dense bricks. To attenuate this effect, directly visible bricks are set to highest rendering priority by defining their accumulated density to be $0$. Since non-dense bricks contain less data, they probably will be rendered a lot faster than bricks with high density. It is important to propagate brick densities along *shortest paths to the camera node* to guarantee a density sorting along lines of sight, since otherwise small densities of bricks may arbitrarily propagate through the graph and lead to incorrect brick sorting. With the correct sorting, less dense bricks will be rendered in correct front-to-back order and will thus lead to rendering of directly visible and relevant features of the data set - under the assumption that irregular and less dense regions of the data set contribute more interesting features than regular and more dense regions. Bricks behind very dense regions of the data set will be sorted very last, which very closely approximates occlusion culling. A good reason to draw dense bricks with less priority is that they can be substituted by impostors more easily, while less dense bricks might contain spatial structures of particles which are harder to detect and approximate by proxy-geometry or other means.

Attempts to generalize the occlusion graph computation to other spatial data structures like kd-trees were made, but without a highly organized spatial structure like given by a uniform grid, correctly deriving occlusion relations between different parts of the data structure remained fruitless.

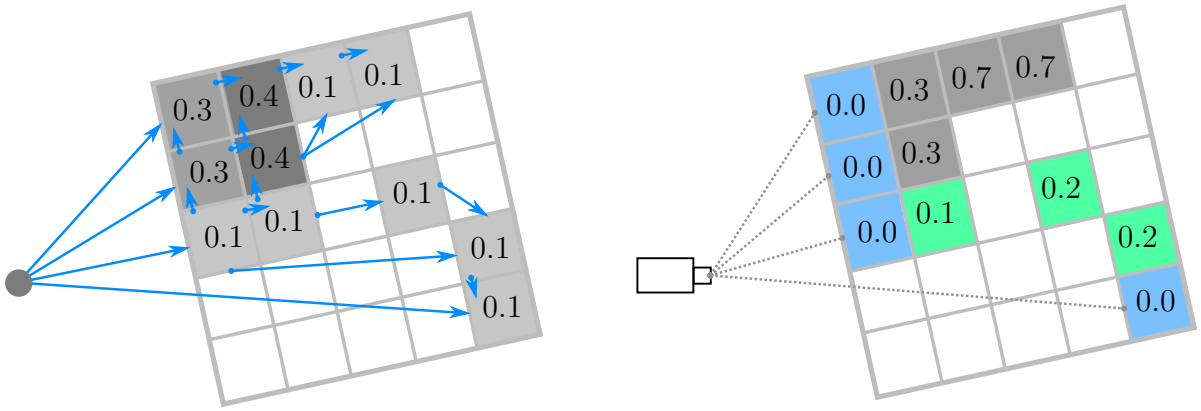



**Figure 3.8:** Density-based brick sorting using the occlusion graph. Left: Assuming the brick densities and occlusion relations as depicted, brick densities are accumulated along shortest paths from the camera node. Right: Resulting accumulated brick densities and rendering order. Directly visible bricks are assigned density $0$ and therefore still rendered first. The other bricks can be roughly partitioned into two sets: bricks with less accumulated density (in green) are going to be drawn before bricks with higher accumulated density (in gray).

# 4 Interactive Rendering of Large Particle Data

In the course of this work an experimental OpenGL rendering framework has been designed and implemented. This was done to rapidly prototype and evaluate the concepts presented in Chapter 2 and Chapter 3, as well as to investigate requirements of an interactive, multi-threaded renderer which needs to handle large data sets. The implementation was done using modern *C++14* and *OpenGL 4.5* with a multi-platform and hardware-independent usability in mind. While the rendering framework is not directly part of the research question, parts of the renderer design and the resulting rendering behavior influence the results of this work. For these reasons, I consider it worthwhile to not only describe rendering algorithms used for particle rendering (as explained in Chapter 2), but to also describe how they interact with the rendering framework to accomplish the visualization of large particle data sets. The assumption made about the particle data is that the data set fits into main memory (and possibly completely fills it), but the data is by far too large to fit into GPU memory. The particle data is to be rendered as is, without altering or reducing the data.

In this chapter, the design of an interactive OpenGL renderer for visualization of large particle data sets is discussed. The renderer provides high interactivity in so far as it handles user input and resulting camera changes with higher priority than full rendering of scene data. This design choice prefers user interactivity over rendering completeness, an approach that is central to the concept of *frameless rendering* studied by [BFMZ94; DWWL05]. After user interaction, the rendered image is progressively refined by streaming and rendering visible scene contents to the GPU. Progressive refinement is a standard technique used for updating rendering contents based on newly generated data, e.g. illumination computations [BFGS86; WLWD03]. In the context of this work, progressive rendering circumvents the issue of data locality on the GPU, i.e. the particle data is streamed to the GPU for rendering bit by bit, while the user may interact with the application (e.g. manipulate the camera) at any time. Data is streamed to the GPU according to the visibility sorting of bricks as described in Section 3.2. A cache of visible particles is built and constantly updated on GPU memory during data streaming. To provide the user with instant visual feedback, this particle cache is rendered as first

approximation of the new scene configuration upon user interaction. Similar render caches have been discussed by [RSH01; WDG02; WDP99] for ray-based rendering.

This combination of rendering techniques trades image completeness for user-interactivity, while attempting to provide the user with a fast-converging rendering (using appropriate visibility sorting) and instant feedback on interaction (using the on-GPU particle cache).

# 4.1 OpenGL Renderer Design

## 4.1.1 Data Loading

Particle data is loaded from MMPLD (MegaMol™ Particle List Data) files and kept in system memory at all times. For dynamic data sets, one time step is loaded at a time, under the assumption that each time step may already completely fill system memory. While loading, the MMPLD data is normalized to a rendering format which holds a position and sphere radius for each particle. Because the data set may occupy all system memory, one buffer for the normalized rendering format is allocated for the data set, and MMPLD file contents are loaded in chunks from storage and written into the final buffer in normalized form. Should several time steps fit into system memory, the system can be easily modified to load time steps in batches. To actually render dynamic data sets, an out-of-core streaming mechanism would be required, which assures timely availability of the next time steps to the renderer. The framework was designed to allow such streaming of data, but time did not permit a final implementation of this subsystem.

Every time step of particle data is sorted into a grid according to the algorithm detailed in Section 3.1.1. The desired grid dimension is provided as user input. The grid construction needs three iterations over the whole data set: (1) count particles per brick, (2) partition particles into their bricks, (3) compute meta data per brick. The first and the last step are done on multiple threads, but for large data sets (1*GiB* and more) this preprocessing takes several seconds up to minutes. This preprocessing will be a limiting factor for rendering of dynamic data sets, if the particle data is not already pre-sorted and provided with corresponding grid meta data. After grid construction, the data set is handled in terms of bricks for the rest of the rendering process.

## 4.1.2 Multi-Threaded Rendering

Consider Listing 4.1 as an example for a usual, single-threaded rendering loop. User interaction is realized by fetching new input events from the operating system and updating the view and projection matrix for the rendering accordingly. Using the new camera pose, visible geometry for the next frame can be determined. The rendering of the final image then depends on the new matrices and the visible geometry. The usual goal of achieving high frame rates in interactive rendering leads to optimizations regarding the `render()` function, such that after quickly rendering the current state, the system can process the latest user inputs, and thus allow real-time interaction with the rendering application. However, if the `render()` function takes too long, many user inputs will stay unprocessed by the system, e.g. the camera matrix will not be regularly

**Listing 4.1** Single-threaded render loop.

```
while(runApplication) {
      UserEvents newUserInput = pollUserEvents();
      Matrices newMatrices = computeViewAndProjectionMatrices(newUserInput);
      GeometryList visibleGeometry = computeVisibleGeometry(newMatrices);
      render(visibleGeometry, newMatrices);
      swapBuffers();
}
```

**Listing 4.2** Multi-threaded render loop.

```
FrameData gpuFrameData, newFrameData; // holds all state of what to render at a time
bool hasNewFrameData = false;

/* on CPU thread */
while(runApplication) {
      UserEvents newUserInput = pollUserEvents();
      if(somethingChanged(newUserInput)) {
            Matrices  newMatrices = computeViewAndProjectionMatrices(newUserInput);
            GeometryList visibleGeometry = computeVisibleGeometry(newMatrices);
            newFrameData = makeNewFrameData(newMatrices, visibleGeometry);
            hasNewFrameData = true;
      }
}

/* on GPU thread */
while(runApplication) {
      if(hasNewFrameData) {
            gpuFrameData = newFrameData;
            hasNewFrameData = false;
      }
      render(gpuFrameData);
      swapBuffers();
}
```

updated with new user interaction. For this reason, and for other reasons regarding the resulting software architecture, it is desirable to decouple the rendering function from the rest of the system and to run it in its own rendering thread, the GPU thread.

Listing 4.2 shows a simplified multi-threaded render loop as used in the application, in which the GPU thread gets notified by the CPU thread when rendering state should change. All OpenGL function calls (and thereby all rendering) are carried out on the dedicated rendering thread, which is fed data updates by the CPU thread via *frame data*. Frame data is simply a bundle of all states relevant for rendering, e.g. view and projection matrices, list of objects to be drawn, options for rendering behavior, etc. This simplifies synchronization between the CPU and GPU thread by enforcing

exactly one interface of communication between the threads. At the same time, this structure makes it more difficult to give feedback from the GPU thread to the CPU thread, e.g. statistics about drawn objects like occlusion queries. For this reason, rendering decisions should be mostly done by the CPU-side. The rendering performance of the multi-threaded render loop (i.e. frame rate) is still dependent on the `render()` function, but if computation of a frame takes a long time, the next frame is going to be rendered according to the most recent user interaction and visible geometry list.

The renderer implementation developed during this work utilizes a multi-threaded render loop as described above. Upon user interaction, the CPU thread performs calculation of new camera and projection matrices, frustum culling of bricks (as in Section 3.1.2), occlusion graph construction and brick visibility sorting by graph traversal (as in Section 3.2). The GPU thread is then provided a *frame data* update consisting of the new matrices and the sorted list of bricks for rendering. The bricks hold pointers to the memory where the respective particle data resides, so no actual copying of particles is done during frame data update.

### 4.1.3 Particle Rendering

Since the particle data is assumed to barely fit into main memory, it is not feasible to allocate GPU storage to hold the whole data set. For this reason, particle data is streamed to GPU memory, as is explained in more detail in Section 4.2. This data streaming essentially is a mechanism to fill an OpenGL buffer of fixed size with particle data for one time use.

Given an OpenGL Buffer Object filled with particle data, the contents are drawn as spheres according to Section 2.1. Depending on the radius of particles, only drawing spheres leads to visual artifacts in regions where the particle's screen footprint is roughly the size of a pixel or smaller.

This is due to the projected size of the sphere's proxy geometry, followed by an under-sampling of the geometry during rasterization, i.e. the geometry is too small to induce fragments. On a large scale, this leads to loss of details in regions with interesting features, as well as loss of perceived spatial density and coherence in distant and dense regions of the data set, as can be seen in Figure 4.1. To counteract this effect, particles are first drawn as spheres, then in a second draw call the same particles are rendered as *GL_POINTS* with size of one pixel. Thus, the second draw call fills regions where spheres were too small to leave a screen footprint, but a large part of the point-particle's fragments will be discarded by the OpenGL pipeline where spheres have already been drawn. Whether this combined *spheres plus points* rendering suits analysis of data sets is up to experts interpreting the data. Either of the render passes can be deactivated if

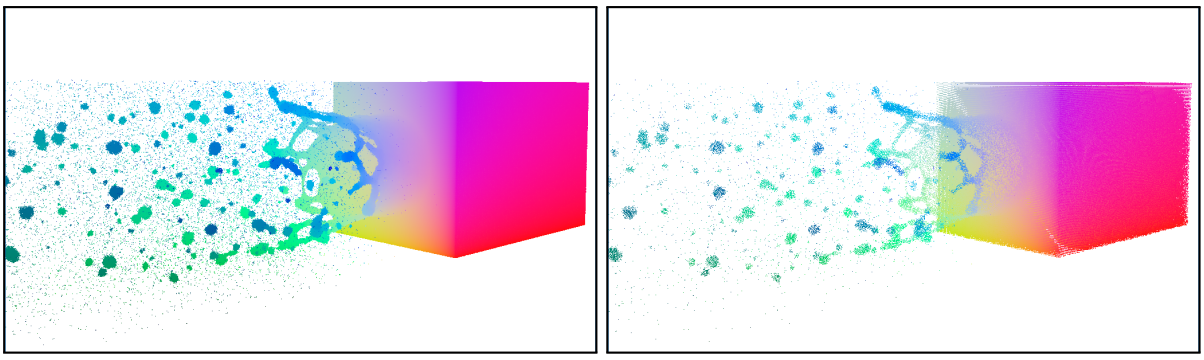


**Figure 4.1:** Comparison of points-only and spheres-only rendering of particles on a laser ablation data set with 48 million particles. Left: Rendering points gives a better overall impression of the cluster-structures in the data by approximating connected surfaces, even in non-dense areas. Right: Rendering only spheres results in loss of perceived large-scale structures, as proxy geometry for particles at the size of individual pixels or smaller does not get rasterized.

it does not help data analysis. From stand point of the rendering application, the two rendering passes only put more workload on the GPU, which will give more conservative performance measures of the rendering algorithms. Put simply, if it will be fast with two geometry passes, it will probably be even faster with only one pass.

One could avoid the second draw pass in one of two ways: either optimizing the vertex shaders to guarantee a rasterization of proxy geometry even when small, or compute particle screen footprint before rendering and drawing points only when necessary. Adjusting vertex shader transformations was attempted, but the approach had several drawbacks. First, the shader code for particle rendering was written to be highly configurable via preprocessor-directives for the purpose of flexibility. Including handling of further edge-cases for correct projection of proxy geometry at pixel size made the code less stable and almost incomprehensible for further development. Secondly, even with the supposedly corrected pixel-size-projection of proxy geometry, particles would still sometimes not be rendered correctly.

The second approach of only issuing *GL_POINTS* draw calls for bricks with small particle screen footprint is feasible as long as particles reside in one OpenGL buffer where their belonging to their respective brick is known. This means that two lists of draw calls are prepared for the renderer. One list with memory ranges into the particle buffer where particles shall be rendered as spheres, and a second list for particles to be rendered as points. Apart from the fact that the computation of those draw-call lists takes a little time, this approach turned out to drastically complicate the code for streaming particles. While the simple streaming variant needs to only copy brick-data into OpenGL buffers and then commit them to rendering, keeping track of particle screen footprint sizes

**Listing 4.3** G-buffer structure for deferred rendering.

```
struct GBufferEntry {
      float3 particlePosition_world;
      float particleRadius;
      float3 proxyGeometryPosition_view;
      byte4 particleColorRGBA;
      /* ... further particle attributes ... */
};
```

per streamed brick and accordingly preparing draw-call lists would have unnecessarily further complicated the streaming code.

Draw-call lists are provided to OpenGL in form of an array of draw-structs, where each array entry defines a range of to-be-drawn vertices into a vertex buffer. A pointer to the draw-array is passed to OpenGL by calling `glMultiDrawArraysIndirect`, which interprets further draw calls from the passed memory, such that no further individual draw calls must be made by the application. Tests with this concept suggested the following. When the particles in *Buffer Object* memory are partitioned in groups, but those groups are arbitrarily spread in the buffer, it is faster to use draw-call lists to issue draw calls of particle groups, such that the particle groups get drawn in front-to-back order. However, if the particles in the buffer are already roughly sorted in front-to-back order (as defined by their brick's visibility sorting), it is faster to draw the contents of the buffer with a single draw call. This probably is related to rendering-order guarantees the OpenGL pipeline makes for geometry drawn by subsequent draw calls [Wik17b], as well as additional overhead for processing of multiple implicit draw calls compared to one draw call. Since the streaming of particle data is done according to the brick's visibility sorting, draw-call lists have no advantage in that use-case.

In summary, to get a consistent rendering of particles without artifacts, issuing a second draw call for particles as *GL_POINTS* turned out to be the least hassle for maintainability of shader and rendering code, and regarding the particle streaming process.

Particles are rendered into an OpenGL *Framebuffer Object (FBO)* for the purpose of deferred shading [ST90] and for the streaming and particle cache algorithms to work. The G-buffer is composed of color attachments of the FBO such that for each screen pixel, information according to the $C-$structure in Listing 4.3 is contained. The suffixes *_world* and *_view* indicate the coordinate system the respective 3d-vector is represented in. The view space position of the initially drawn proxy geometry is saved for each particle to later reconstruct the original rays used for ray casting, which eliminates numerical errors due to possible ray direction variations during deferred shading. Important attributes of the particle are the particle position and radius, furthermore a color for the particle is saved. During shading, the sphere's surface-normal can be reconstructed from the ray intersection with the spherical particle.

The final rendering of particles consists of two geometry passes into the G-buffer FBO (spheres draw and points draw), followed by a shading pass of the particles into the system framebuffer.

## 4.2 Progressive Rendering and Data Streaming

In this section, streaming and progressive rendering of *static* particle data is considered. For *dynamic* data sets, largely the same arguments and considerations as for static data apply. For dynamic data consisting of several time steps, the key requirement is assumed to be a periodic switching of underlying particle data. This change of particle data can be handled like an automatically generated user event, implying a new set of *frame data* for the *rendering state*, where the renderer is provided the next particle time step as geometry instead the current particle time step. However, the basic streaming and rendering mechanisms will be the same for static and dynamic data, as the GPU thread only receives pointers to particle buffers for rendering, and is thus unaware of the concept of a particle data time step.

The GPU thread receives a visibility-sorted list of bricks from the CPU thread, accompanied by the current view- and projection-matrix resulting from user interactions. Due to the multi-threaded render loop, this frame data update to the render thread happens asynchronously. Depending on the load on the GPU, the threads may produce and consume new frame data at different rates. As long as the user does not make further inputs, the rendering application works with the most recent input state, i.e. the rendering works on static data inputs (matrices and geometry) for a given time period until new user interactions require an updated image. Given a set of static geometry to be rendered, rendering software usually draws all geometry at once and the resulting image is presented to the user only after all rendering operations finished.

The multi-threaded render loop already decouples user interactions and geometry preprocessing from rendering performance. To further improve interactivity and to allow rendering of arbitrarily large data sets, the `render()` function itself must utilize a geometry streaming and drawing algorithm. This rendering procedure must be able to be aborted at any time, such that new frame data for rendering is processed with minimal latency after it was announced by the CPU thread. Simultaneously, newly streamed and drawn particle data should be made visible to the viewer as quickly as possible, which means that the streamed rendering should swap front and back buffers often. This leads to a progressive rendering procedure where the rendered image of the data set gets drawn and presented to the user bit by bit. For this reason, important features and directly visible parts of the data set should be rendered first to give an immediate impression of the data set. The occlusion graph and the derived visibility sorting in Section 3.2 have been studied for this exact purpose.

Because the arbitrarily large data set is to be visualized to the viewer without any modification or omission of data, there seems to be no other sensible approach to handle large data than gradually streaming geometry and progressively refining the rendered image - while enabling the user to interact with the scene at any time, at the cost of

initiating the streaming process anew for each new camera pose. With such a streaming process in place, there is also little motivation to employ strategies like occlusion culling to reduce geometry load on the GPU. If geometry can be determined to be important for the rendered image, it should definitely be streamed and rendered with highest priority and as fast as possible. If geometry can be proven to not contribute to the image at all (e.g. via frustum culling), it can be safely omitted from rendering. However, if geometry does not contribute to the image and gets correctly visibility-sorted and streamed very last, this does not affect image quality or interactivity in a negative way.

The data streaming to GPU-memory is implemented using a dedicated streaming-thread. The streaming-thread copies particle data to preallocated GPU memory and notifies the render thread of newly available data. After rendering the data and assuring that the memory resources are not used by OpenGL anymore, the GPU thread handles the memory resources back to the streaming-thread. The data copy mechanism is outsourced to the streaming-thread so the render loop can quickly move on to swap the framebuffer after drawing newly available particles. Meanwhile, new particle data can be copied to OpenGL buffers by the streaming-thread without delaying OpenGL operations in the main rendering procedure.

The interaction between GPU- and streaming-thread is implemented by a producer-consumer resource-ring, into which produced resources (OpenGL buffers filled with particle data) get inserted by the streaming-thread. The GPU thread takes (fetches) produced resources out of the ring, consumes them by rendering the buffer's contents, and commits them back into the resource-ring as empty (unproduced) resources. The streaming-thread can then fill the resources with particles, etc. pp. To guarantee a finite time for rendering resource buffers, the GPU thread fetches and renders produced resource buffers only once per render loop iteration. The streaming-thread greedily fills the OpenGL buffers (resource buffers) with particle data as defined by the brick's visibility sorting, i.e. the sorted list of bricks is traversed front-to-back and the brick's particles are copied into OpenGL memory in that order. After the streaming-thread filled all momentarily available resource buffers and committed them for rendering, it sleeps until it is signaled the availability of new empty resources. When all bricks' contents have been processed by the streaming-thread, it sleeps until user interactions trigger a new frame data state, and thus initiate the streaming of a new list of bricks. The *NGSphereRenderer* module of the MegaMol™ [GKM+15] framework implements a similar streaming process for particle data, but in a single-threaded and non-interruptible fashion where the whole data set gets streamed and drawn before a swapping of front and back buffer occurs.

The OpenGL buffers are allocated with a fixed size (for example, 2 million particle entries per buffer) and then the streaming-thread fills up a buffer, that buffer is immediately committed for consumption, and particle copying is continued into the next empty

buffer. This streaming process may split one brick's particle contents into several resource buffers, or several bricks may fit into one resource buffer. The number of resource buffers in the resource ring, as well as the individual size of the buffers, can be altered during runtime, which may be a way to dynamically optimize performance of the streaming process. New resource buffers can be allocated and added to the resource-ring during run time to improve streaming performance, as long as the GPU has enough video memory to provide newly allocated memory. The size of the resource buffers themselves should be chosen such that the GPU can render that amount of particles comfortably, i.e. further progress of the renderer should not be held back when drawing one resource buffer full of particles.

The resource buffers are allocated as *OpenGL Buffer Objects* by the GPU thread and their memory is *mapped persistently* (i.e. a pointer to the internal OpenGL memory is retrieved). This implies that the underlying OpenGL memory is of type *Storage*, which can not be re-allocated with a new size after initialization. The mapped OpenGL memory pointer allows the streaming-thread to write data to OpenGL memory without issuing OpenGL function calls. This mechanism is important, since OpenGL functions may only be called from one thread at a time, namely the thread in which the OpenGL context is currently active. Repeatedly switching the OpenGL context between GPU- and streaming-thread for purpose of copying data would not be reasonable.

For the time streaming occurs on GPU-side, the CPU thread must assume that particle buffer contents must not be altered. This is especially important for handling dynamic data sets, where the buffer of the last time step may be used to load particle data for the next time step. To give such a guarantee, the streaming-thread aborts all copy operations upon a frame data update from the CPU thread. Streaming of particle data is immediately started for the newly provided bricks list. The CPU thread locks execution until the streaming-thread acknowledges receiving the new frame data, after which it does not read memory referenced by the previous bricks list anymore.

On the GPU thread, the incoming resource buffers with new particle data are constantly drawn into the G-buffer for deferred shading. The G-buffer gets only cleared after the view or projection matrix changes, otherwise it stays uncleared and receives rendered particles through the streamed rendering process. In each render loop iteration, the shading pass uses the current G-buffer state to draw particles to the system framebuffer, which gets shown to the user after the following buffer swap. The system framebuffer is cleared after every render loop iteration to allow correct drawing of further elements, like a user interface or additional geometry besides particles.

With this rendering procedure, the frame rate as defined by *completed frames per second* becomes an inappropriate metric for measuring rendering performance. A more suitable measure is how fast the image converges towards the final rendering, and how well the first few frames after user interaction approximate the scenes contents.

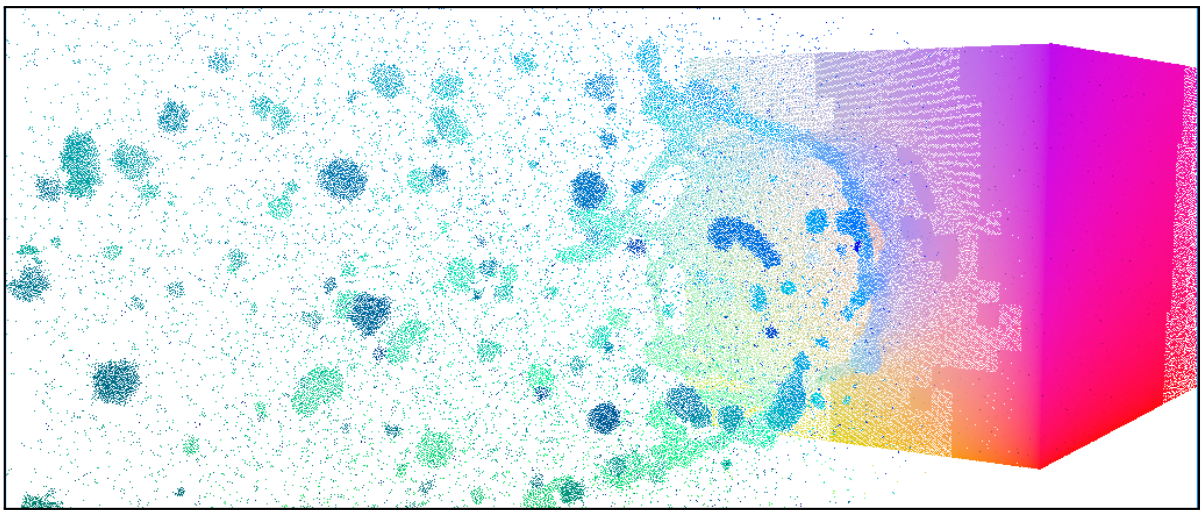


**Figure 4.2:** Effect of the on-GPU particle cache during user interaction. During user actions, the particle cache ensures comfortable interactivity by providing an instant feedback about the impact of user inputs on scene geometry. The dense part of the data set on the right has already been streamed to the GPU by the progressive streaming and rendering algorithm, while the non-dense appearing parts of the rest of the data set have not been streamed yet and are drawn from the particle cache. Note that the particle cache does not explicitly keep track of which parts of the data set (e.g. individual bricks) are important to keep in GPU memory, but it derives visible (and therefore important) scene geometry from G-buffer contents. Without the particle cache, the user would only see the freshly streamed particles during user interaction, which produces unpleasant flickering and gives no useful feedback about how the camera pose changes relative to scene contents.

## 4.3 Utilizing a GPU Particle Cache

Due to the deferred shading chosen for rendering, all currently visible particles have a footprint in the G-buffer. Considering again the information saved in the G-buffer, you notice that the G-buffer not merely holds some abstract information about scene geometry, but that it already contains the actual visible scene geometry. Listing 4.4 highlights this by explicitly splitting particle data from the rest of the G-buffer's content. When user interaction leads to new view or projection matrices (i.e. to a new *view-projection matrix*), the whole scene must be rendered again with the new camera configuration. Thus, the whole particle data set must be streamed to the GPU again. However, using the G-buffer contents from the last frame before matrices changed, we have a good approximation of probably visible geometry which already resides in GPU memory. So for the purpose of giving the user the best possible first frame after user

**Listing 4.4** G-buffer structure holding particle data.

```
struct Particle {
      float3 particlePosition;
      float particleRadius;
      byte4 particleColorRGBA;
      /* ... further particle attributes ... */
};

struct GBufferEntry {
      Particle particle_world;
      float3 proxyGeometryPosition_view;
};
```

interaction, the G-buffer can be used as an on-GPU cache of to-be-rendered particles upon change of the view-projection matrix. The positive effect of the particle cache during user interaction is illustrated in Figure 4.2.

In order to use the particles contained in the G-buffer, they must be extracted from the G-buffer's textures and made available for rendering. This is done by a screen-space *particle reduce pass* which copies particle data from the G-buffer into an *OpenGL Shader Storage Buffer (SSBO)* that will act as particle cache. The maximum number of particles visible on the screen at a time is limited by the current framebuffer resolution. Particles drawn as spheres will likely occupy more than just one pixel. So the screen resolution is a conservative estimate for the amount of particle data the SSBO particle cache needs to contain. For a rendering resolution of *4K*, assuming a particle structure as defined in Listing 4.4, the buffer size of the SSBO will be

$$3840 \times 2160 \times 20\,bytes = 8294400 \times 20\,bytes \approx 158\text{MiB}$$

which is a reasonable amount of memory that can be expected to be available on modern GPUs. However, simply copying the particle data from the G-buffer into the SSBO would result in many duplicate particles, as individual particles may leave a large screen footprint when rendered, and thus many G-buffer entries in a neighborhood will contain the same particle, as depicted in Figure 4.3. So the G-buffer contains connected regions of different sizes which represent the same particle. The connected regions representing individual particles should be reduced or somehow processed such that only one pixel of a region writes its particle data into the particle cache.

The *particle reduce pass* proceeds as follows. In the manner of deferred shading, a screen-filling quad is drawn which generates one fragment shader invocation for each screen pixel. Each fragment shader invocation reads particle data from the G-buffer corresponding to its screen position. A preallocated particle array of fixed size is bound as SSBO to the shader, so each fragment can write into it, furthermore a shared *atomic counter* with initial value zero is accessible by each fragment. Depending on the G-buffer

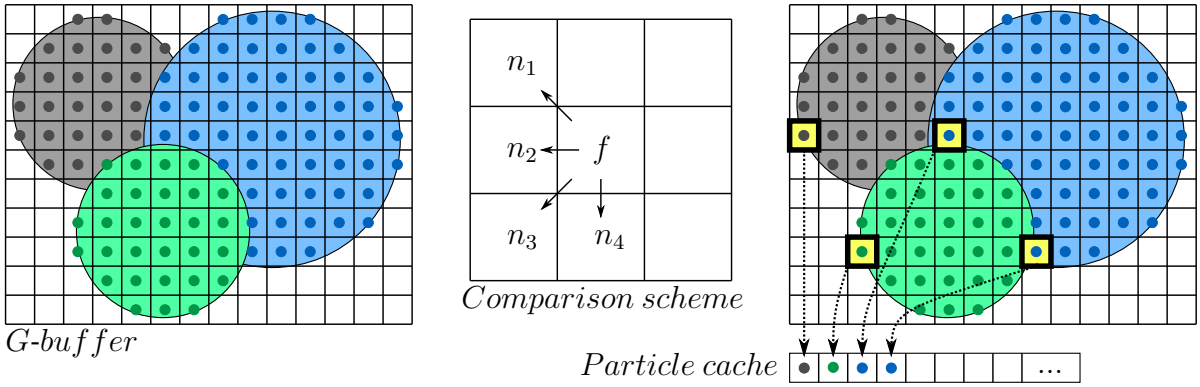



**Figure 4.3:** Particle cache filling scheme. Left: Connected regions in the G-buffer representing rendered particles. Here, screen footprints of three particles are shown. Middle: Decision scheme for fragment $f$ whether to write its own contents into the particle cache. If any of the neighbor fragments $n_1, n_2, n_3, n_4$ has the same G-buffer entries as $f$, the current fragment $f$ will discard itself and not write to the particle cache. Right: Sparse particle cache writes (bold pixels in yellow) resulting from the decision scheme. Some particles may be written several times to the cache (blue region on the right). Still, many unnecessary cache writes are avoided.

entries of itself and its local neighborhood, each fragment decides whether to write its own particle data into the SSBO. The atomic counter assigns each fragment an individual index, which the fragment uses to write into the particle cache array bound by the SSBO.

Each fragment compares its own G-buffer entry with the entries of its left and lower neighbors, as depicted in Figure 4.3. If any neighbor contains the exact same data, the fragment will discard itself. If none of the neighbors contain the same data, the fragment will increment the atomic counter and use the previous counter value as index into the SSBO particle cache data array. OpenGL guarantees that no fragment which *discards* itself will issue atomic counter or SSBO manipulations, which is the key property to guarantee a sparse filling of the particle cache [Wik16]. To assure correct handling of particles at the image edges, G-buffer color textures are configured to handle out-of-bounds accesses by *GL_CLAMP_TO_BORDER*, and the border value is set to $(0.0f, 0.0f, 0.0f, 0.0f)$ as RGBA values. Note that the OpenGL texture border is not the outermost line of pixels of the texture (those are the edge pixels), but the border is a color assigned to pixel reads which read beyond the texture dimensions. The border entry is chosen as a particle with zero radius, under the assumption that no particle in the data set will have zero radius.

After the *particle reduce pass*, the SSBO holding the particle cache can be used as geometry-input for drawing the contained particles. For rendering, the particle cache buffer is handled like all other buffers and needs no special treatment. Conveniently, the atomic buffer used by the *particle reduce pass* holds the number of particles in the particle cache, and this value can directly be used to issue draw calls of correct size. When the GPU thread receives an updated view-projection matrix it clears the G-buffer for streaming of new particles and directly draws the current particle cache with the updated matrix as first approximation of scene contents. The G-buffer entries are reduced into the particle cache after each draw call issuing particles, meaning the SSBO gets completely rewritten after every draw call. Still, this procedure is fast enough to not affect rendering performance negatively. The overhead introduced by the particle reduce pass seems to be in the range of $1 - 4$ milliseconds, depending on the capability of the GPU and the current window resolution. Especially for large data sets, the main workload during rendering is still caused by the large amount of particle data. In comparison, the rendering of the particle cache itself is quite fast, since the cache holds only a few million particles, with the very conservative upper limit of 8.3 million particles for a 4K resolution window. The particle cache usually will hold multiple entries of the same particle due to the used screen-space decision scheme. One could further condense the particle cache by finding and eliminating recurring particles, e.g. by traversing the particle cache in compute shaders and marking recurring particles for deletion, followed by some cleanup-mechanism. Such approaches were not investigated further, since the rendering of the particle cache did not influence overall rendering performance on any of the GPUs the application was tested on.

For dynamic data sets, it is clear that the particle cache can not be used as an approximation for the next time step. Drawing the particle cache when updating the time step would mix up particle data from the previous time step with the new one, which rarely will be a desired effect. The concept of the particle cache may however help to choose which particles of the next time step should be rendered first. This can be done if every particle in the data set has an individual identification number (ID), which is valid over all time steps. This ID can be tracked during the rendering process and retrieved from the particle cache upon switching of time steps. The ID then allows to find and render the seen particles in the new time step data. The particle cache can further be used to implement a z-prepass to pre-initialize the depth buffer without writing color buffers before actually rendering geometry with a new view-projection matrix. Such a z-prepass turns out to impact performance neither positively nor negatively. Moreover, the z-prepass introduces an unnecessary rendering of particles which are going to be visible anyway. So although a z-prepass is fast on modern GPUs and may improve performance, for this specific scenario directly rendering the contents of the particle cache is sufficient.

On small changes of the camera pose, the particle cache allows an immediate rendering of relevant scene contents. Particles which are far away from the camera and only occupy one pixel tend to disappear from the particle cache after camera motion, as illustrated in Figure 4.2 at the beginning of this section. This happens because small particles previously projected into separate pixels are projected into the same pixel after camera motion. If camera motions reveal previously unseen parts of the scene, the particle cache can not replace a fast streaming of newly visible geometry. However, previously visible particles act as a point of reference during camera movements, which greatly improves interactivity and scene perception.

# 5 Results

In this chapter, performance measurements of the rendering application described in Chapter 4 are presented. To evaluate interactivity, the relevant CPU-side and GPU-side steps for frame data update are measured. Those are frustum culling, occlusion graph construction and traversal, and brick-sorting on the CPU, and the latency to acknowledge and start rendering a new frame data state on the GPU. Furthermore, three different visibility sorting methods and resulting progressive rendering behavior are evaluated.

The measurements were conducted on the **TITAN** system, which is a Windows 10 desktop PC with Intel Core i7-3820 4-core processor @ 3.6GHz, 32GB DDR3 memory @ 1600MHz and a GeForce GTX TITAN with 6GB GDDR5 video memory, running the official NVIDIA driver version 376.53. The program is compiled on Visual Studio Enterprise 2015 in Release mode. The application is evaluated with static particle and point cloud data. The data sets used for evaluation are two particle data sets resulting from laser ablation simulations, **LaserBig** and **LaserCharge**, and a point cloud resulting from the scanning of an industry hall, **Halle10**. The data sets have the following properties

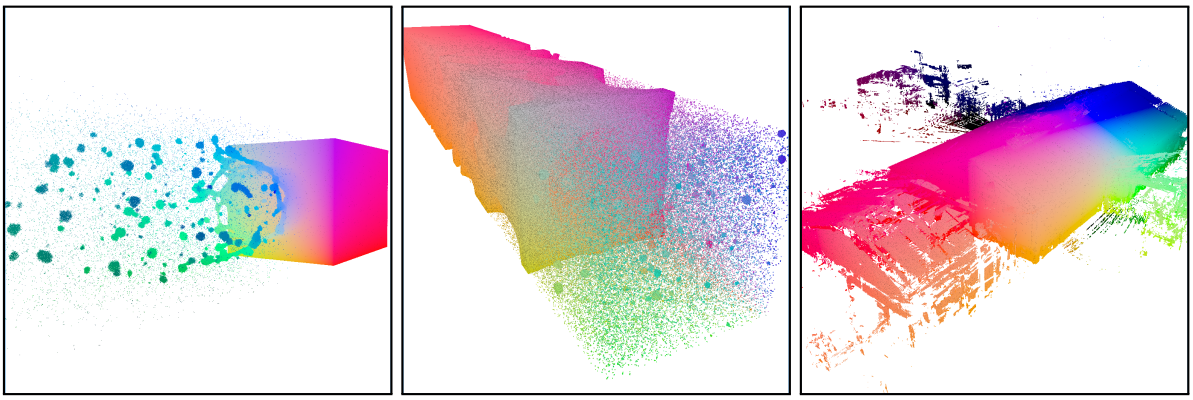


**Figure 5.1:** Data sets used for evaluation. All three data sets can be explored interactively with the developed rendering application, with representation of the complete data set and little preprocessing time. Left: *LaserBig* data set with 48 million particles (733MB). Middle: *LaserCharge* data set with 199 million particles (3.2GB). Right: *Halle10* data set with 1.5 billion points (24GB).

- **LaserBig** scene. Data set consisting of *48 million* particles, *733MB* in size. The preprocessing time to sort the particles into a uniform grid is between 6.6 seconds (building a $25 \times 5 \times 5$ grid) and 10 seconds ($200 \times 37 \times 37$ grid).
- **LaserCharge** scene. Data set consisting of *199 million* particles, *3.2GB* in size. The preprocessing time to sort the particles into a uniform grid is between 25 seconds ($25 \times 5 \times 5$ grid) and 51 seconds ($200 \times 33 \times 33$ grid).
- **Halle10** scene. Point cloud data set with roughly *1.5 billion* points, *24GB* in size. Sorting the points into a grid was measured to take between 3 minutes 22 seconds ($25 \times 23 \times 10$ grid) and 6 minutes 16 seconds ($200 \times 184 \times 80$ grid)[1].

While the *LaserBig* scene fits into GPU memory and can be brute-force-rendered with 12fps (as spheres) to 23fps (as points) on the TITAN, the *LaserCharge* scene renders with roughly 2fps when rendered in a brute-force manner. The *Halle10* data set can be loaded into system memory, but does not fit by any means into GPU resources. The developed rendering application applies a simple Lambertian reflectance model to shade spheres of rendered particles. The particles are assigned a color according to their position in space. Renderings of the three data sets are shown in Figure 5.1. For each data set, an overview of the scene is rendered where all particles are inside the view frustum. The LaserBig scene consists of a dense block of particles from which individual particles and larger clusters of particles break away. The LaserCharge scene shows a similar characteristic, but furthermore the block of particles exhibits internal structural deformations and bubble-formation. The Halle10 scene is a point cloud of an industrial building, approximating scanned surfaces of the building itself and contained machinery and equipment. In the Halle10 scene, a lot of scattered data-points and seemingly wrongly assembled sub-scans of the building can be spotted (see right image in Figure 5.1).

[1] For comparison, building a pkd-tree for 134 million particles (2.1GB) takes roughly 8 minutes, when finding the median value for splitting by explicitly sorting the data along one axis in each recursion step.

## 5.1 Evaluating Interactivity

To maintain interactivity, the acceptable latency between user input and system response is roughly $100$ milliseconds [CRM91; Mil68; Mye85]. According to the multi-threaded render-loop, user-interactivity is restricted by the time period it takes the application to process user input, during which the application can not receive and process new user interactions. This input-processing latency is determined by the CPU-side processing time for the operations culling, occlusion graph processing and bricks sorting, and by the GPU-side time to acknowledge newly received frame state data and particle data, after which no data for the previous frame state is rendered and no previous particles resources are read by the streaming thread. The CPU-side processing time depends on the dimensions of the grid enclosing the data set, which defines the number of bricks to be processed by culling, occlusion computation and visibility sorting. The GPU-side depends on the number of particles rendered in each frame, which is defined by the maximum number of resource buffers in the multi-threaded resource-streaming mechanism and the amount of particles each resource buffer can contain. Upon user interaction, the streaming-thread guarantees to the CPU-thread that no old particle resources will be used after acknowledging the new frame data state. Because the streaming-thread occasionally needs to wait for availability of empty resource buffers, the responsiveness of the frame data acknowledgment depends on the ability of the GPU-thread to render all filled resource buffers containing particles, and committing the empty resources back to the streaming-thread.

Table 5.1 shows maximum frame-data update measurements for the three data sets under different configurations of grid size and GPU streaming. This way, interactivity with the system is measured by the maximum latency the system has between user inputs. The CPU-side measurements show computation times in milliseconds for frustum culling of bricks, Occlusion Graph building time and visibility sorting, dependent on different grid sizes and occupied bricks in the grid. The GPU-side times show frame data acknowledgment time in milliseconds depending on the resource ring size, which is the number of resource buffers used for streaming. The size of each individual resource buffer was set to 2 million particles. During measurements, the camera positions as in Figure 5.1 were used for the data sets, so all bricks were inside the view frustum and added to computation times. The window resolution was set to $1024^2$. For each of the measured frame update steps, the maximum value out of a series of consecutive frame updates are reported. The frame updates were triggered by user interactions lasting several seconds. Note that the CPU-side and GPU-side frame update mechanisms are independent from each other, so the different grid sizes (longest dimension $50$ up to $200$) and ring sizes ($3$ up to $15$) were not measured together, as the table might falsely suggest. Instead, each combination of CPU-side and GPU-side table entries is a valid

configuration of the renderer, which would result in a maximum frame update time that is the sum of the two separate configurations.

Depending on the number of occupied bricks, the CPU-side update can take only a few milliseconds or up to $200ms$, as can be seen in the sum-columns of the tables in Table 5.1. The culling algorithm adds least to the frame update, with a maximum computation time of $17ms$ for processing 196 thousand bricks. Like the culling, building the occlusion graph and deriving the visibility sorting scale with the number of bricks. Due to issues with the multi-threaded occlusion graph implementation, the numbers presented are for the single-threaded computation of occlusion graph edges. The visibility sorting, which traverses the graph edges and assigns levels or accumulated densities to graph nodes and sorts the nodes according to the assigned metric, also utilizes a single thread.

Building the occlusion graph takes the longest time during frame update, with up to $133ms$ for computing occlusion edges in a $200 \times 33 \times 33$ grid with $196,626$ occupied bricks for the *LaserCharge* scene. Building an occlusion graph for the largest tested grid on the *Halle10* scene takes almost the same amount of time ($126ms$), but for only a quarter of occupied bricks ($54,972$). This is explained by the densely filled grid for the LaserCharge where occlusion-ray traversal through the grid immediately terminates at a neighbor brick, while the Halle10 scene exhibits a lot of empty grid cells due to a lot of scattered data-points in all directions which enlarge the bounding box for the data set. The empty space traversal through the sparsely occupied grid then leads to a similar computational effort for building the occlusion graph as for the densely occupied grid. In both cases, computing the graph leads to an overall frame update time that is going to take longer than the targeted $100ms$, which will reduce interactivity for the user. The visibility sorting takes up to $50ms$ in the worst case and scales with the number of occupied bricks, rather than their positioning in space.

The GPU-side time to finish rendering particles of the old frame and acknowledge the new frame scales with the number of rendered particles coming from the resource-ring buffers, which is just the usual time it takes to render $(\#ResourceBuffers) \times (\#ParticlesPerBuffer)$ particles. For most frame updates after user interaction, during measurements the time to acknowledge the new frame took zero or only very few milliseconds, but once in a while, acknowledging the frame took a very long time of up to $278ms$. This long time comes from the GPU-thread rendering all available resource buffers at once and waiting for the rendering to finish until releasing the resource buffers again, and the resulting noticeable input latency is exactly the latency usually introduced by slow rendering and resulting low frames per second.

So all in all, the latency of the application to respond to user inputs scales with number of bricks in the scene and maximum possible GPU-load for one frame. By adjusting the grid size or optimizing the CPU-side algorithms for preprocessing, or adjusting GPU-side workload by changing the number and size of resource buffers, interactivity of the

application can be adapted to the capabilities of the used hardware and size of particle data set. For grid sizes leading to less than $100,000$ occupied bricks, and for resource ring sizes of 3 to 5 resource buffers each holding 2 million particles, the TITAN system can visualize the tested data sets interactively.

| **LaserBig** | | | | | | |
|---|---|---|---|---|---|---|
| | CPU-side | | | | GPU-side | |
| # Occupied Bricks (Grid Size) | Culling | Graph Build | Visibility Sort | Sum | Ring Size | Acknowledge |
| 3,518 ($50 \times 10 \times 10$) | 0 | 2 | 0 | 2 | 3 | 31 |
| 18,973 ($100 \times 19 \times 19$) | 1 | 16 | 4 | 21 | 5 | 59 |
| 45,781 ($150 \times 28 \times 28$) | 4 | 40 | 11 | 55 | 10 | 128 |
| 91,865 ($200 \times 37 \times 37$) | 7 | 71 | 21 | 99 | 15 | 198 |

| **LaserCharge** | | | | | | |
|---|---|---|---|---|---|---|
| | CPU-side | | | | GPU-side | |
| # Occupied Bricks (Grid Size) | Culling | Graph Build | Visibility Sort | Sum | Ring Size | Acknowledge |
| 4,023 ($50 \times 9 \times 9$) | 0 | 3 | 0 | 3 | 3 | 42 |
| 28,232 ($100 \times 17 \times 17$) | 2 | 23 | 6 | 31 | 5 | 83 |
| 89,027 ($150 \times 25 \times 25$) | 7 | 65 | 23 | 95 | 10 | 198 |
| 196,626 ($200 \times 33 \times 33$) | 17 | 133 | 50 | 200 | 15 | 278 |

| **Halle10** | | | | | | |
|---|---|---|---|---|---|---|
| | CPU-side | | | | GPU-side | |
| # Occupied Bricks (Grid Size) | Culling | Graph Build | Visibility Sort | Sum | Ring Size | Acknowledge |
| 2,790 ($50 \times 46 \times 20$) | 0 | 4 | 0 | 4 | 3 | 24 |
| 12,316 ($100 \times 92 \times 40$) | 1 | 19 | 2 | 22 | 5 | 44 |
| 30,265 ($150 \times 138 \times 60$) | 3 | 56 | 8 | 67 | 10 | 121 |
| 54,972 ($200 \times 33 \times 33$) | 7 | 126 | 11 | 144 | 15 | 191 |

**Table 5.1:** Maximum times in milliseconds for frame update procedures after user interaction for different data sets, grid sizes and streaming configurations. The CPU-side operations measured are culling, occlusion graph construction and visibility sorting. For the GPU-side, maximum time until acknowledging and starting to render a new frame data state is measured, depending on number of resource buffers used for streaming.

## 5.2 Evaluating Progressive Rendering and Visibility Sorting

The progressive rendering gradually building up the image depends on the speed of the streaming mechanism and the visibility sorting imposed on the bricks inside the view frustum. While the streaming itself can not be arbitrarily optimized, since the actual data transfer to the GPU is handled by the GPU driver and limited by hardware capabilities, the order in which bricks get drawn determines how quickly the user can perceive important structures and interesting features of the data set, and make decisions to further navigate through the data.

To evaluate the quality of the developed occlusion-graph-based visibility sortings, the convergence towards the finished rendering is analyzed. Three methods of sorting the bricks (*visibility sorting*) for rendering are considered:

- Draw by **Depth**. Sort the bricks front-to-back by distance to the camera, according to the $z$-coordinate of the bricks' center in view space.
- Draw by **Level**. Use the bricks' occlusion level according to the level assignment resulting from traversal of the occlusion graph. Draw small levels before higher ones, sort bricks of the same level by their distance to the camera (front-to-back).
- Draw by **Density**. Based on the traversal of the occlusion graph, use the accumulated density along the shortest path between camera-node and brick-node for sorting. Draw bricks with smaller density first.

The data sets were each streamed with the three *visibility sortings* and the framebuffer status was saved to disk after each draw call that rendered new particles to the image. For each data set, one **Overview** streaming-images series and one **Detail** series was recorded, with the Overview showing the whole data set and the Detail showing a close-up of part a of the scene. For each series of images, the convergence towards the last image is considered. For this, each image of the series is compared to the converged image by computing the mean-squared error (MSE) between the two images. To visualize the convergence, this error is plotted against the number of the image in the series, resulting in the graph plots in Figure 5.2, Figure 5.3 and Figure 5.4. For a visibility sorting, it is desirable to have a good impression of scene contents fast, and a quick convergence towards the final image. Thus, the thing to look for in the plots of streaming convergence is a rapid fall-off of the graph at the start of the streaming (*good first impression*) and quick reaching of the $x$-axis (*zero difference to final reference image*). The $y$-axis of the plots does not directly show the MSE of images, but the square root of the MSE. This is done in order to bring the $y$-values of the plotted graphs on a scale where they can be more easily compared. Otherwise, the interesting regions of streaming-begin and streaming-end are harder to compare because the actual MSE values are widely distributed over a very large range.

Figure 5.2, Figure 5.3 and Figure 5.4 not only show graph plots of the visibility sorting convergences, but also show selected images during streaming for each visibility sorting, as well as the reference image. Note that for all shown and compared images, for each scenario individually, the same number of particles has been drawn across all visibility sortings to achieve the rendered image - it is the different prioritization of different bricks in the scene that leads to the different image outputs. Additionally, for each data set statistics are presented for completely streaming the Overview and Detail scene view. The statistics show the amount of streamed memory, number of streamed particles, the time it took to stream and render that amount, and the number of buffer swaps of the system framebuffer, as well as the number of draw calls it took to render all particles (one draw call per filled resource buffer). The number of buffer swaps reveals how often the user could have interacted with the application during streaming and gotten a timely response, while the number of draw calls represents the number of image updates where new particles were presented to the user. For all measurements, the streaming was done with a resource ring of size 5, each resource buffer holding a maximum of 2 million particles. The grid size was chosen to have dimension $100$ along the longest data domain extent. The window resolution for rendering was $1024^2$.

The graph plots in Figure 5.2, Figure 5.3 and Figure 5.4 suggest that the Density sorted rendering is the best choice for most data sets and camera configurations. Out of the six analyzed rendering situations, only the *Detail* view of the interior of the *Halle10* scene shows a significant advantage for the Depth and Level visibility sorting. In that streaming sequence, the Density sorting delays rendering of a central part of the image (a large block of machinery with supposedly many data points). Furthermore, in the Halle10 Detail view the Density sorting renders parts of the scene which later will be overdrawn by other geometry (top right corner of Density sorting screenshot), which shows that the density propagation in the occlusion graph can even go around dense areas and prioritize less dense areas partially covered by other geometry. This could be a bug in the graph traversal and density-accumulation implementation, or this specific configuration of camera pose and partitioning of scene data into bricks, combined with a lot of empty space in the grid, triggers this special case of sorting. Furthermore, the exploration of the Halle10 data set feels more natural when contents closer to the camera are perceived first, rather than patches of surfaces getting randomly updated somewhere in the image, as seen in the screenshot for the Density sorting. This is due to the long streaming and rendering time of several seconds even for the Detail view of Halle10, where the rendering converges after about $1.3$ seconds (convergence after $85$ draw calls in relation to $207$ total draw calls in $2.9$ seconds) after progressively rendering $6.3$GB of point data. For comparison to that overall streaming time, the images from the streaming process for the Halle10 Detail view are for the 10th draw call, which made the depicted scene approximation visible to the user after only $0.14$ seconds, a tenth of the total time until the image finally converged, and only a twentieth of the total processing

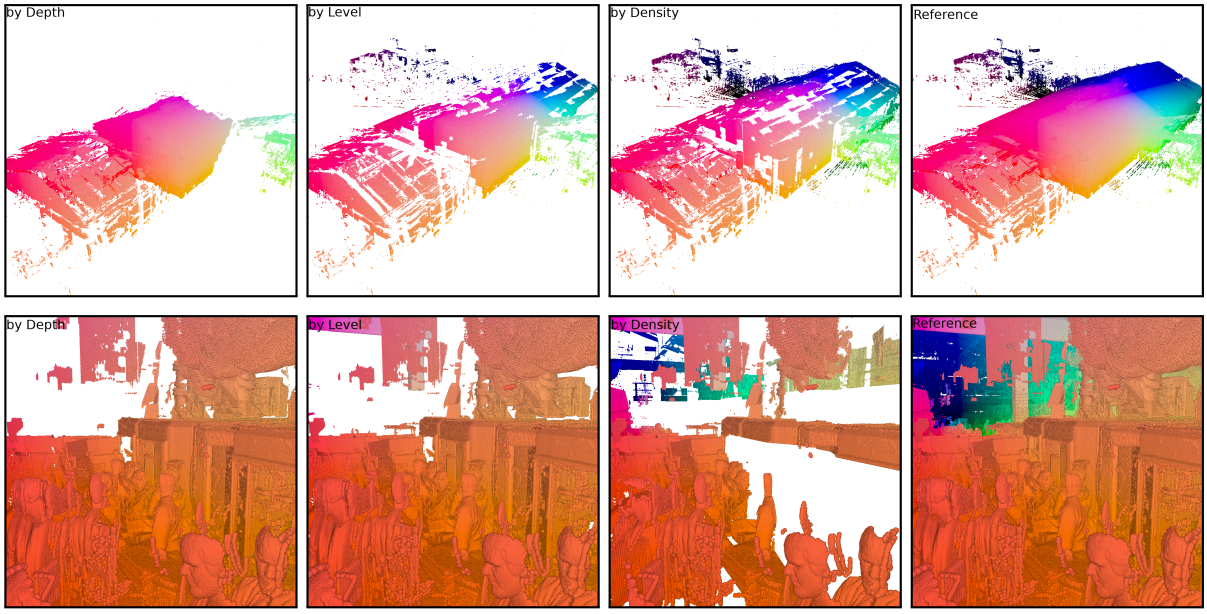
time a brute-force rendering of that amount of data would take. During the complete streaming of the Halle10 Overview and Detail scenes, the framebuffer is swapped every $11$ milliseconds (total streaming time divided by number of framebuffer swaps), which is the latency with which the GPU-thread would have acknowledged and started rendering a new frame data state resulting from user interaction. Thus, from perspective of the GPU-side, interactivity is ensured to the user for this large data set.

The *LaserBig* scene is small enough that with the Density sorting, the progressive rendering gives a very good first impression on the first frames and converges after $3-4$ draw calls. Depending on the camera pose, the Depth and Level sorting converge notably slower (for the Overview), or almost as fast as the Density sorting, when the camera is positioned in a way such that all three visibility sortings imply roughly the same sorting for bricks (for the Detail view). In the Detial view, the Density sorting does really well in presenting the user with a very good first impression of the data set with the first drawn frame. Although the whole data set needs $24$ draw calls to have been streamed and rendered, the images resulting from the density sorting converge after $4-6$ draw calls, which translates to a finished image after $58-87$ milliseconds of progressive rendering. During streaming, the GPU thread swaps framebuffers every $11$ milliseconds on average, which assures user interactivity and a quick presentation of rendering progress.

The *LaserBig* Overview and Detail analysis again shows the good quality of the Density sorting. For this particularly elongated particle data domain that fills the grid very densely, the nature of the Level sorting is beautifully illustrated in the Overview streaming plot. Because the data set gets rendered layer by layer according to the occlusion graph levels, and because the bricks of each layer closest to the camera get drawn first (which for this particular view is the non-dense part of the data set), with each newly drawn occlusion-level layer of the grid, the plot for the level sorting drastically reduces the mean-squared error to the reference image and then stays level while all other brick of the same occlusion-graph-level further behind in the data set get drawn. This leads to a step-like plot of the convergence, because bricks farther away do not contribute to the image as that image area has already been initialized with a good approximation. Like for the other tested data sets, user interactivity (7ms up to $18$ms between buffer swaps) and fast image convergence (only the first half of all draw calls contributes to the image) show good results.

In conclusion, the developed occlusion-graph based Density sorting gives overall good results and for many scenarios it performs better than rendering bricks by Depth or by occlusion graph Level. The employed progressive streaming and rendering is able to ensure interactive exploration of data sets consisting of up to $1.5$ billion particles. By utilizing the developed visibility sorting strategies, a good first impression of new

scene views and a fast-converging rendering towards the actual image of the data is achieved.




| **Streaming Statistics** | Memory | Time [ms] | #BufferSwaps | #DrawCalls | #Particles |
|---|---|---|---|---|---|
| Overview | 23.9GB | 11,018 | 960 | 785 | 1,563M |
| Detail | 6.3GB | 2,906 | 272 | 207 | 413M |

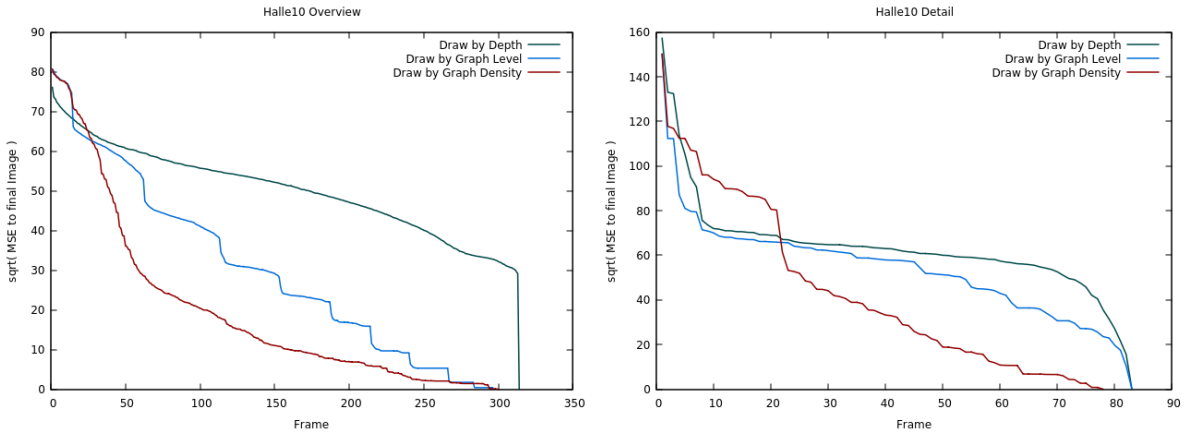
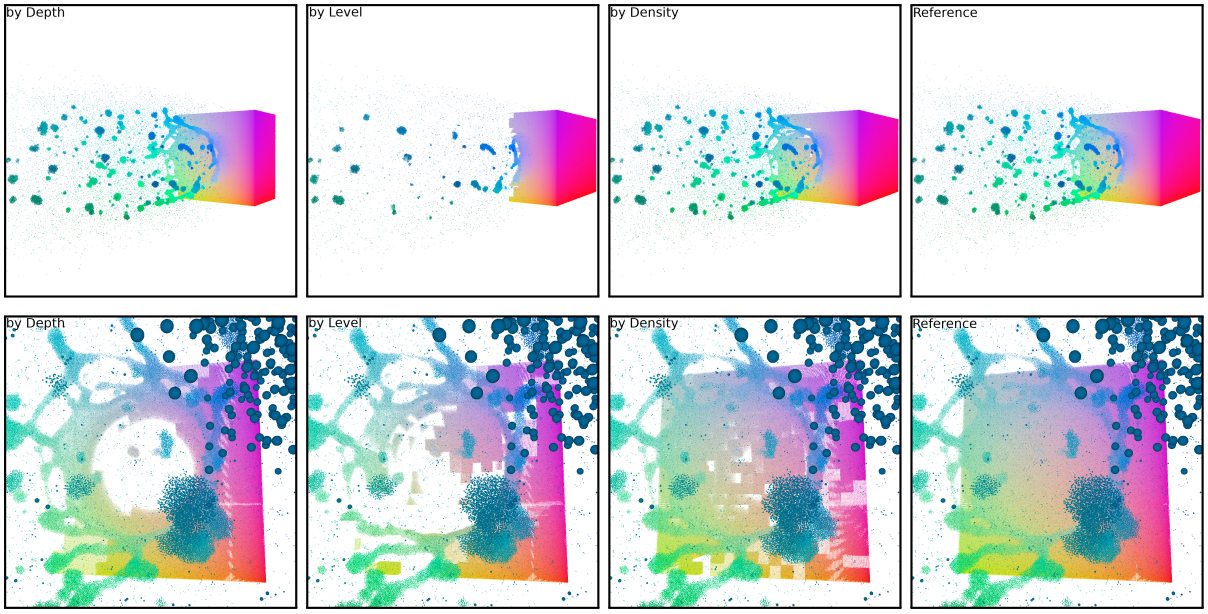



**Figure 5.2: Images**: Top Images: Overview view, frame number 50 for each visibility sorting (Left to right: Depth sorted, Level sorted, Density sorted) and the converged image (rightmost). The Density sorting gives the best impression of the final image at this early stage of streaming. Bottom Images: The same for frame number 10 of the streaming for the Detail view. Here, Depth and Level sortings give the better early impression of the data set, because the Density sorting leaves a central part of the image empty for a long period during streaming. **Table**: Statistics of the streaming for Overview and Detail views. The streaming and rendering of the complete 23.9GB data set takes 11 seconds. **Plots**: Streaming-convergence of the three visibility sortings towards the final image of the Overview view (left plot) and the Detail view (right plot).




| Streaming Statistics | Memory | Time [ms] | #BufferSwaps | #DrawCalls | #Particles |
|---|---|---|---|---|---|
| Overview | 733MB | 349 | 29 | 24 | 47.9M |
| Detail | 731MB | 347 | 31 | 24 | 47.9M |

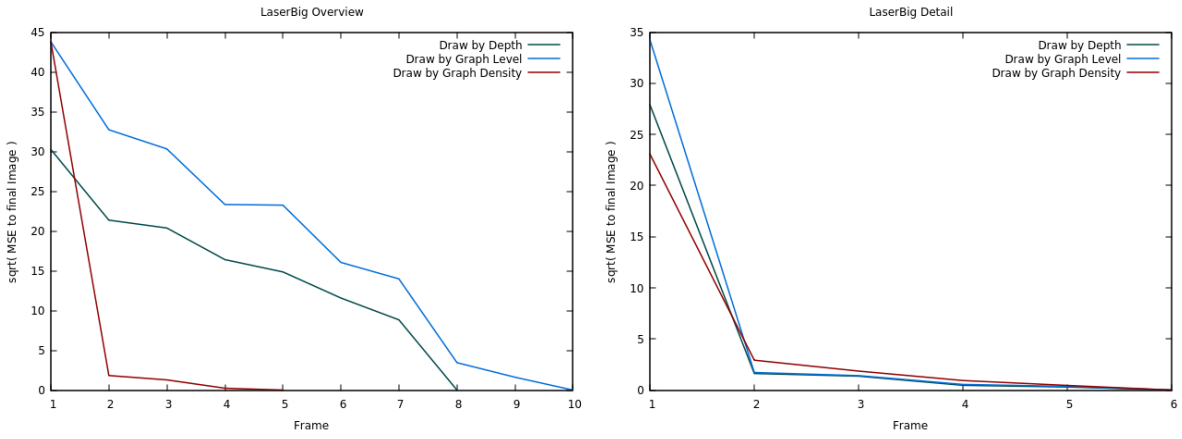
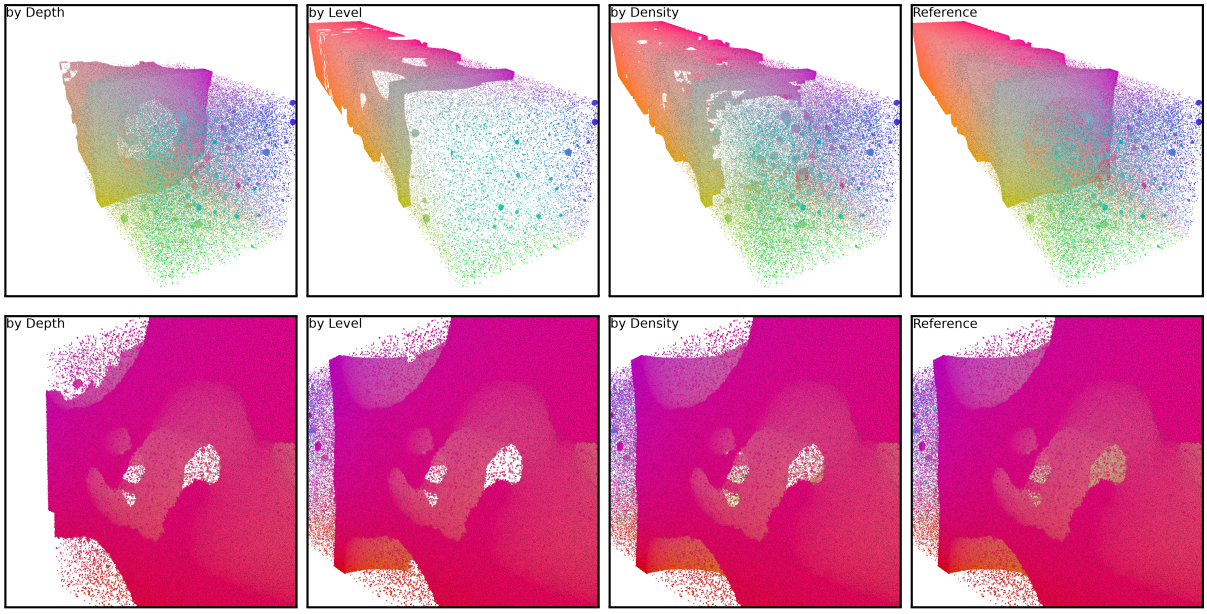



**Figure 5.3: Images**: Top Images: Overview view, frame number 2 for each visibility sorting (Left to right: Depth, Level, Density) and the converged image (rightmost). The Density sorting gives the best first impression. Bottom Images: The same for frame number 1 of the streaming for the Detail view. Here too, the Density sorting gives the best first impression. **Table**: Statistics of the streaming for Overview and Detail views. The streaming and rendering of the whole data set takes roughly 350 milliseconds, which suggests that without the streaming overhead, the data set could probably be rendered fairly fast in a brute-force way. **Plots**: Streaming-convergence of the three visibility sortings towards the final image of the Overview view (left plot) and the Detail view (right plot). The Overview view converges in only five frames for the Density sorting. For the Detail view, all methods converge similarly because the camera pose implies almost the same sorting of bricks for all visibility sortings.




| Streaming Statistics | Memory | Time [ms] | #BufferSwaps | #DrawCalls | #Particles |
|---|---|---|---|---|---|
| Overview | 3GB | 1619 | 93 | 101 | 199.8M |
| Detail | 364MB | 219 | 34 | 12 | 23.9M |

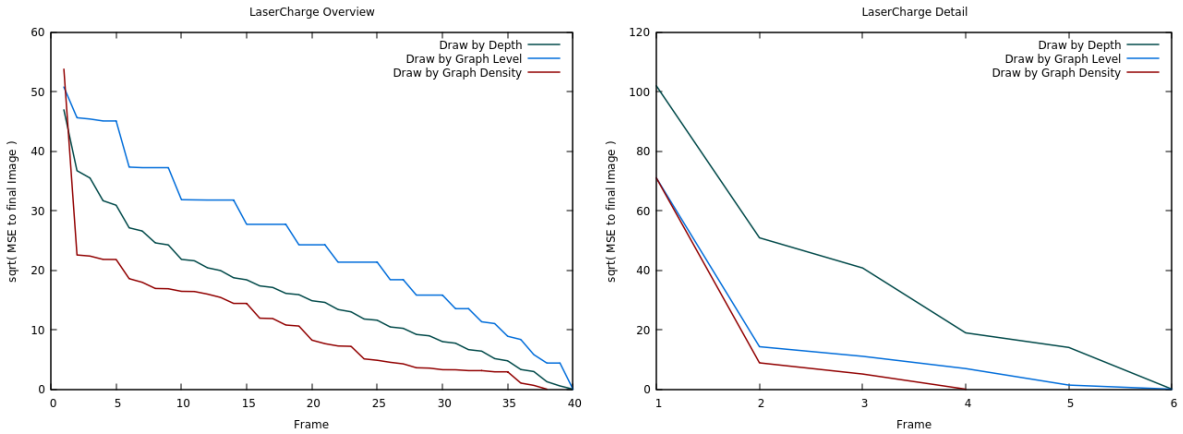



**Figure 5.4: Images**: Top Images: Overview view, frame number 2 for each visibility sorting. Bottom Images: Also frame number 2, for the Detail view. **Table**: Statistics of the streaming for Overview and Detail views. **Plots**: Streaming-convergence of the three visibility sortings. For the Overview view, the rapid fall-off of the Density sorting at the beginning of streaming is notable. This suggests that the Density sorting works well on data sets with lots of structure resulting from density changes. The Level sorting gives a bad first impression because it can not quickly fill the right lower image region where most of the non-dense region of the data set resides. The step-like fall-offs of the Level sorting result from brick layers of the non-dense region being drawn and improving image convergence, while drawing brick layers of the same level in the occluded data set region further behind does not improve image quality. For the Detail view, the Density sorting again converges fastest.

## 5.3 Evaluating Particle Cache Impact

The particle cache provides an important, instant feedback about camera pose changes during user interaction by re-projecting previously seen particles into the next rendered frame. This way, the particle cache provides an approximation of scene contents as point of reference during user movements. Without such a mechanism, during interaction the user would be presented with a flickering image of only partially rendered parts of the scene. Moreover, because user interactions lead to a new camera pose every $100ms$ or so, in that short time the progressive rendering and streaming will not have streamed the whole data set to the GPU. Thus, the particle cache is really crucial for a smooth user experience.

Despite its importance, I could not come up with a quick, meaningful way to measure the impact of the cache on user interactivity. The ability of the particle cache to provide a useful approximation of scene contents depends on the nature of user input (camera rotation or movement, direction of movement), as well as speed of camera movement and duration of interaction. If the user very quickly rotates or moves the camera in a direction far outside the current view, no particle information about that part of the scene resides in the particle cache yet. Particle data previously out of view needs to be streamed to the GPU according to the previously described and evaluated progressive streaming and rendering methods. As long as user interaction not not too fast and mostly looks at previously seen geometry, the particle cache provides an enjoyable navigation through the scene. Because the particle cache is extremely cheap to render and to build, it shows no substantial GPU-overhead on powerful GPUs like the TITAN or Radeon R9 used for testing. Even on the humble integrated mobile graphics chip of my personal laptop (Radeon HD 8330 with an AMD A4-5000 4-core processor @ 1.5GHz), the deduction and constant re-rendering of particle cache contents takes roughly $1 - 5$ milliseconds per frame, which is a negligible processing overhead when interactively rendering particle data sets with sizes of up to $100$ million particles (which can be done with the developed rendering application on my modest laptop).

To provide an idea of the look and feel of the particle cache during interaction, in Figure 5.5, Figure 5.6 and Figure 5.7 I include screenshots of the Halle10 and LaserBig data sets during user interaction, taken on the TITAN system.

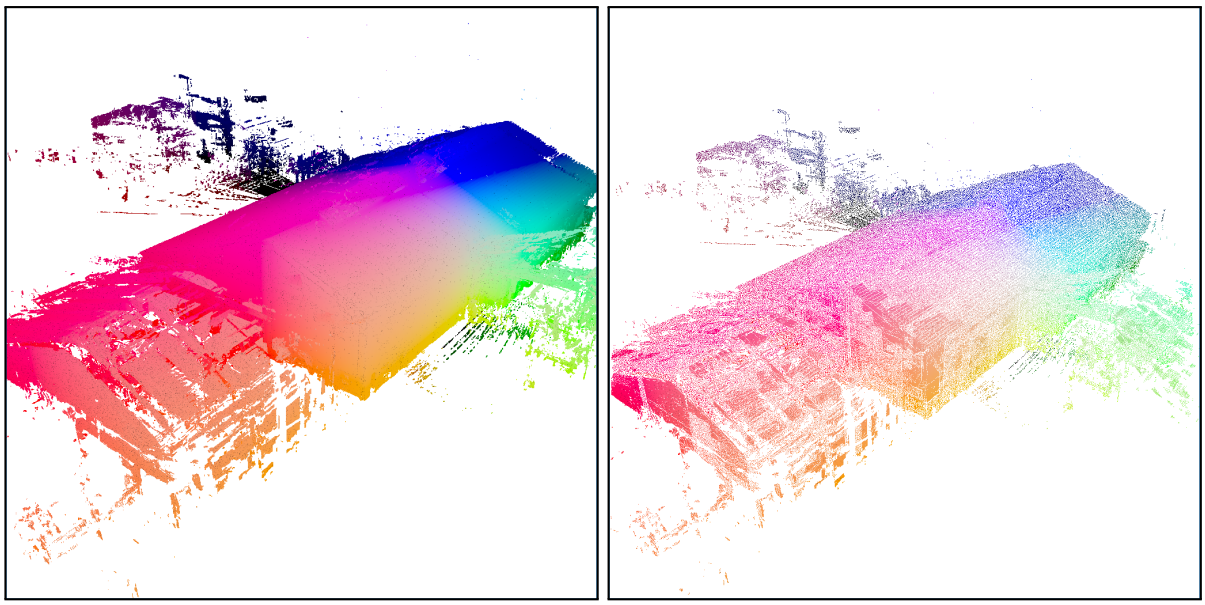


**Figure 5.5:** Converged Overview rendering of the 1.5 billion points Halle10 scene on the left. On the right, first frame after small user movement towards the data set. Since the Halle10 data set takes about 1.3 seconds to converge to the image on the left, usually the user would have to wait for that long until seeing a rendered image displaying the new camera pose. With the particle cache, an immediate feedback is possible. The appearance of smoothly-connected surfaces in the left picture is lost after user interaction. This is due to the user movement towards the data set, which spreads visible particles further apart and reveals previously hidden parts of the scene. Furthermore, when very small particles occupying different pixels get re-projected, they can get projected to the same pixel after camera motion, thus z-culling all particles falling into that pixel except for one.

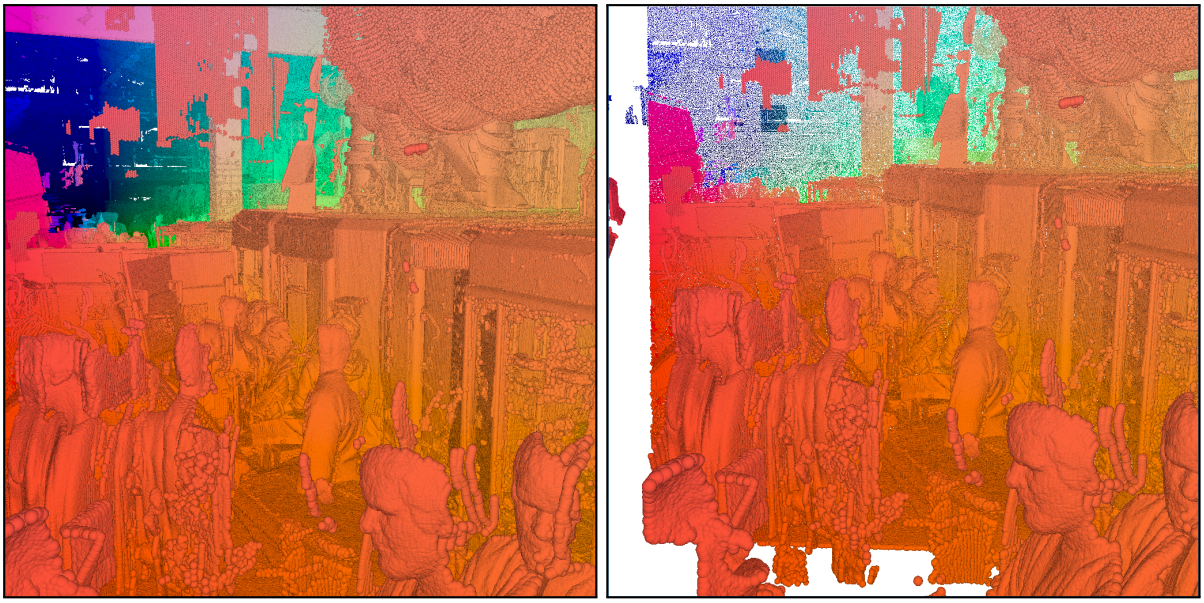


**Figure 5.6:** Fully converged Detail view of the interior of the Halle10 scene on the left. On the right, the camera made a rapid rotation towards the left lower corner. Particle cache contents provide a smooth image transition while previously unseen scene contents get streamed for rendering (white empty area on the left and bottom). In the back of the particle cache image (upper left corner), particles previously occupying few pixels get thinned, while particles drawn as larger spheres in the front retain a smooth image.

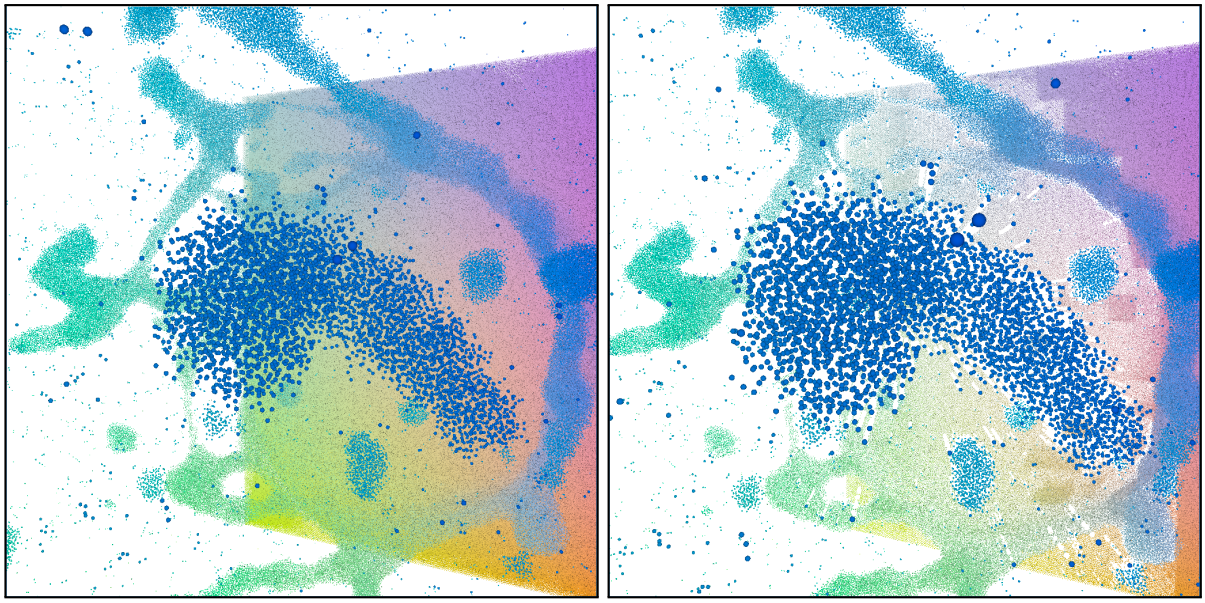


**Figure 5.7:** Converged Detail rendering of the LaserBig scene on the left. On the right, image of the scene after a movement towards the central particle cluster. The spheres of the central particle cluster move towards the user during camera motion, erasing particles further behind from the particle cache during consecutive frames. The erasure appears because the large spheres occlude small particles further behind during movement, thus erasing them from the cache. Nonetheless, the overall structure of the data set is largely preserved during camera motion.

# 6 Conclusion and Future Work

In this work, the interactive visualization of large particle data has been studied. An interactive rendering application has been designed and implemented, which allows to interactively explore particle data sets limited in size only by system memory. To realize this application, several methods regarding rendering techniques and data organization have been designed and evaluated.

Two techniques of particle rendering in OpenGL have been evaluated, the rasterization-oriented approach of rendering proxy geometry and ray casting spheres, and the on-GPU traversal of a hierarchical spatial data structure - the pkd-tree. The performance characteristics of sphere ray casting depend on several factors like implementation details and used graphics hardware, and this approach is largely geometry-bound for brute-force rendering of millions of particles. However, the sphere ray casting is quite reliable, because it uses common OpenGL mechanisms and shows acceptable performance on the tested graphics hardware. The pkd-tree algorithm by [WKJ+15] for rendering large particle data was ported to the GPU and evaluated. The evaluation shows that from the two GPUs (NVIDIA Titan and AMD Radeon R9) this algorithm was tested on, only the AMD Radeon R9 200 could run the algorithm with somewhat good performance. The pkd-tree traversal on the Radeon R9 becomes faster than geometry processing for rasterization for data sets containing 67 million particles and upwards. However, the pkd-tree traversal on the Radeon R9 is not fast enough to be interactive, and the pkd-tree structure used for the implementation has a few drawbacks that discourage using the pkd-tree for rendering of large particle data sets on the GPU. So for rendering particle data on the GPU with OpenGL, rasterizing proxy geometry and ray casting spheres is the best option.

To organize large particle data sets and prepare that data for rendering, a grid-based approach was chosen, where the particles of the data set get sorted into the cells of a uniform grid. The grid cells containing particles, called bricks, serve as basic units for processing scene geometry. The bricks are used to cull particle data outside the view frustum, and the spatial arrangement of the bricks inside the grid is used to compute a visibility sorting of scene contents, based on the occlusion relation between bricks. The occlusion relations between bricks are computed by traversing the uniform grid with rays, and an occlusion graph is built from those occlusion relations. Using the

occlusion graph, the bricks are sorted according to their visibility in the scene, which leads to a sorted list of bricks that is drawn by the rendering algorithms in that order. Two visibility sortings are derived from the occlusion graph, the Level sorting which orders bricks according to their assigned level in a breadth-first traversal of the graph, and the Density sorting which incorporates the amount of particles per brick (the density of a brick) into the brick sorting and can thereby allow to render non-dense parts of the data set quicker, while delaying rendering of dense but less important scene parts. The undertaken measurements of those visibility sortings suggest that the Density sorting indeed leads to a better visibility sorting of scene contents.

An interactive rendering application has been developed in *C++14* and *OpenGL 4.5* and follows a multi-threaded design principle to reduce the processing latency of user inputs wherever possible, thus ensuring interactivity. Additionally, the rendering of huge particle data sets is not done by brute force, but a progressive particle rendering is employed, backed by a multi-threaded streaming mechanism for particle data. User inputs are preprocessed by a CPU-thread, which prepares a visibility-sorted list of bricks for rendering. The sorted list of to-be-drawn bricks is then passed to the GPU- and streaming-thread, which are designed to immediately respond to that new rendering state and render the newly provided particle data progressively. The user is presented with all updates of the progressive rendering immediately and can interact with the system at any time. Each user interaction starts the visibility sorting and progressive rendering process anew for the changed camera state resulting from user inputs. To further improve interactivity and navigation-ability through the data set, an on-GPU particle cache is derived from the most current framebuffer contents and rendered upon user interaction to give an immediate feedback about the effect of user inputs. For small camera pose changes, this has the effect of being able to interactively navigate through the original data set without having to wait until the progressive rendering converges to a final image. Measurements of the multi-threaded rendering application show that depending on certain configurations, user inputs can be successfully processed and realized into rendering state changes with latencies of at most $100ms$, thus ensuring interactivity to the user. Furthermore, the interactivity of the system is parametrized by factors like the grid size used to spatially organize the particles and maximum amount of particle data drawn by the GPU per draw call during progressive rendering, which allows to tune those parameters depending on capabilities of the available hardware.

Using the developed techniques, a particle data set of $1.5$ billion data points (24GB in size) was tested and found to be interactively explorable. By shifting the focus from the usual image-correctness centric approach to a user-interactivity centric and data-management aware rendering application design, backed by algorithms to sort scene data by importance for the rendering and giving the best possible immediate feedback to the user upon interaction, the results of this work allow to interactively explore data sets limited in size only by system memory.

As future work, there are several areas with room for improvement. Algorithmic optimizations may improve brick-processing like frustum culling and the occlusion graph. The frustum culling is not a bottleneck in the evaluation measurements, but implementation of an exact box-test (as opposed to the used sphere-test) would reduce streaming of some unnecessary bricks. A hierarchical culling approach using some octree-like hierarchy would further improve speed of frustum culling. Speeding up the occlusion graph and visibility sorting computations will greatly benefit user interactivity for large grids. Attempts to implement a multi-threaded occlusion graph construction resulted in synchronization issues which lead to false graph construction. Besides multi-threading the current occlusion graph structure, the next algorithmic optimization is to extend the occlusion graph computation to hierarchical spatial structures like octrees or other kinds of grid- and voxel hierarchies. However, loss of the highly regular structure of the grid makes quick occlusion computation between different parts of the spatial structure more difficult. A more direct approach to optimize the existing occlusion graph structure would involve low-level code optimization and utilization of Streaming SIMD Extensions (SSE) of modern CPUs, or using an already highly optimized CPU ray-tracing library like Embree [WWB+14] to compute the occlusion-graph ray casting in the grid. For accelerated occlusion graph traversal and visibility sorting, multi-threaded graph traversal algorithms might improve performance.

To improve perception of the data set during user interactions, simplified and extremely cheap approximation-geometry for scene contents could be drawn for data parts which were not yet streamed to the GPU. This would involve good-quality impostors for individual bricks or groups of bricks which are cheap to render and easy to derive from brick contents during preprocessing. Extending the rendering of particles itself with more involved techniques would be the next step to elevate the quality of produced images, like improving perception of the data set using estimated normals for shading and by employing ambient occlusion [GRDE10; SGG15]. Extending the developed framework itself to support programming of highly configurable graphical user interfaces (GUI) would allow to realize more usability and support features like manipulation and evaluation of transfer functions. However, it turns out that the multi-threaded renderer design makes implementation of GUI capabilities more complicated than usual, because GUI elements might change values on several threads, or several threads might change GUI values. For example, GUI inputs might change values on the CPU thread regarding grid configuration or trigger loading of a new data set, or GUI inputs might change state on the GPU thread regarding rendering options like shading. Implementing such features in a multi-threading-safe way calls for a clean design of framework subsystems like user inputs and GPU-thread synchronization, supported by appropriate multi-threading helper classes.

# Bibliography


[Ahn13] S. H. Ahn. "OpenGL Projection Matrix." In: *URl: http://www. songho. ca/opengl/gl_ projectionmatrix. html* (2013) (cit. on p. 64).

[AM00] U. Assarsson, T. Möller. "Optimized View Frustum Culling Algorithms for Bounding Boxes." In: *J. Graph. Tools* 5.1 (Jan. 2000), pp. 9–22. ISSN: 1086-7651. DOI: 10.1080/10867651.2000.10487517. URL: http://dx.doi.org/10.1080/10867651.2000.10487517 (cit. on p. 63).

[AW87] J. Amanatides, A. Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing." In: *EG 1987-Technical Papers*. Eurographics Association, 1987. DOI: 10.2312/egtp.19871000 (cit. on p. 71).

[BDS05] T. Boubekeur, F. Duguet, C. Schlick. "Rapid Visualization of Large Point-Based Surfaces." In: *EUROGRAPHICS International Symposium on Virtual Reality, Archeology and Cultural Heritage (VAST)*. Ed. by M. Mudge, N. Ryan, R. Scopigno. Eurographics. Pise, Italy: Eurographics, Nov. 2005. DOI: 10.2312/VAST/VAST05/075-082. URL: https://hal.inria.fr/inria-00260886 (cit. on p. 19).

[Ben75] J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching." In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: http://doi.acm.org/10.1145/361002.361007 (cit. on pp. 47, 48).

[BFGS86] L. Bergman, H. Fuchs, E. Grant, S. Spach. "Image Rendering by Adaptive Refinement." In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 29–37. ISBN: 0-89791-196-2. DOI: 10.1145/15922.15889. URL: http://doi.acm.org/10.1145/15922.15889 (cit. on p. 79).

[BFMZ94] G. Bishop, H. Fuchs, L. McMillan, E. J. S. Zagier. "Frameless Rendering: Double Buffering Considered Harmful." In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '94. New York, NY, USA: ACM, 1994, pp. 175–176. ISBN: 0-89791-667-0. DOI: 10.1145/192161.192195. URL: http://doi.acm.org/10.1145/192161.192195 (cit. on p. 79).

[BHZK05] M. Botsch, A. Hornung, M. Zwicker, L. Kobbelt. “High-quality surface splatting on today’s GPUs.” In: *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005*. June 2005, pp. 17–141. DOI: 10.1109/PBG.2005.194059 (cit. on pp. 18, 25).

[BK03] M. Botsch, L. Kobbelt. “High-quality point-based rendering on modern GPUs.” In: *11th Pacific Conference onComputer Graphics and Applications, 2003. Proceedings.* Oct. 2003, pp. 335–343. DOI: 10.1109/PCCGA.2003.1238275 (cit. on p. 18).

[BS02] H. C. Batagelo, W. Shin-Ting. “Dynamic scene occlusion culling using a regular grid.” In: *Proceedings. XV Brazilian Symposium on Computer Graphics and Image Processing*. 2002, pp. 43–50. DOI: 10.1109/SIBGRA.2002.1167122 (cit. on p. 69).

[BW03] J. Bittner, P. Wonka. “Visibility in Computer Graphics.” In: *Environment and Planning B: Planning and Design* 30.5 (2003), pp. 729–755. DOI: 10.1068/b2957. eprint: http://dx.doi.org/10.1068/b2957. URL: http://dx.doi.org/10.1068/b2957 (cit. on p. 67).

[BWPP04] J. Bittner, M. Wimmer, H. Piringer, W. Purgathofer. “Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful.” In: *Computer Graphics Forum* 23.3 (2004), pp. 615–624. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2004.00793.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2004.00793.x (cit. on p. 67).

[Cat74] E. E. Catmull. “A Subdivision Algorithm for Computer Display of Curved Surfaces.” AAI7504786. PhD thesis. 1974 (cit. on p. 67).

[CICS05] S. P. Callahan, M. Ikits, J. L. D. Comba, C. T. Silva. “Hardware-assisted visibility sorting for unstructured volume rendering.” In: *IEEE Transactions on Visualization and Computer Graphics* 11.3 (May 2005), pp. 285–295. ISSN: 1077-2626. DOI: 10.1109/TVCG.2005.46 (cit. on p. 69).

[CMSW04] R. Cook, N. Max, C. T. Silva, P. L. Williams. “Image-space visibility ordering for cell projection volume rendering of unstructured data.” In: *IEEE Transactions on Visualization and Computer Graphics* 10.6 (Nov. 2004), pp. 695–707. ISSN: 1077-2626. DOI: 10.1109/TVCG.2004.45 (cit. on p. 69).

[CRM91] S. K. Card, G. G. Robertson, J. D. Mackinlay. “The Information Visualizer, an Information Workspace.” In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’91. 100 milliseconds 100ms ms. New Orleans, Louisiana, USA: ACM, 1991, pp. 181–186. ISBN: 0-89791-383-3. DOI: 10.1145/108844.108874. URL: http://doi.acm.org/10.1145/108844.108874 (cit. on p. 97).

[Dan02] J. J. Daniel Sykora. *Efficient View Frustum Culling*. Central European Seminar on Computer Graphics. 2002. URL: http://old.cescg.org/CESCG-2002/DSykoraJJelinek/paper.pdf (cit. on p. 66).

[DVS03] C. Dachsbacher, C. Vogelgsang, M. Stamminger. "Sequential Point Trees." In: *ACM SIGGRAPH 2003 Papers*. SIGGRAPH '03. San Diego, California: ACM, 2003, pp. 657–662. ISBN: 1-58113-709-5. DOI: 10.1145/1201775.882321. URL: http://doi.acm.org/10.1145/1201775.882321 (cit. on p. 18).

[DWWL05] A. Dayal, C. Woolley, B. Watson, D. Luebke. "Adaptive Frameless Rendering." In: *Proceedings of the Sixteenth Eurographics Conference on Rendering Techniques*. EGSR '05. Konstanz, Germany: Eurographics Association, 2005, pp. 265–275. ISBN: 3-905673-23-1. DOI: 10.2312/EGWR/EGSR05/265-275. URL: http://dx.doi.org/10.2312/EGWR/EGSR05/265-275 (cit. on p. 79).

[EBN13] J. Elseberg, D. Borrmann, A. Nüchter. "One billion points in the cloud – an octree for efficient processing of 3D laser scans." In: *ISPRS Journal of Photogrammetry and Remote Sensing* 76 (2013). Terrestrial 3D modelling, pp. 76–88. ISSN: 0924-2716. DOI: http://dx.doi.org/10.1016/j.isprsjprs.2012.10.004. URL: http://www.sciencedirect.com/science/article/pii/S0924271612001888 (cit. on pp. 19, 57).

[FGKR16] M. Falk, S. Grottel, M. Krone, G. Reina. "Interactive GPU-based Visualization of Large Dynamic Particle Data." In: *Synthesis Lectures on Visualization* 4.3 (2016), pp. 1–121. DOI: 10.2200/S00731ED1V01Y201608VIS008. eprint: http://dx.doi.org/10.2200/S00731ED1V01Y201608VIS008. URL: http://dx.doi.org/10.2200/S00731ED1V01Y201608VIS008 (cit. on p. 30).

[FKE13] M. Falk, M. Krone, T. Ertl. "Atomistic Visualization of Mesoscopic Whole-Cell Simulations Using Ray-Casted Instancing." In: *Computer Graphics Forum* 32.8 (2013), pp. 195–206. ISSN: 1467-8659. DOI: 10.1111/cgf.12197. URL: http://dx.doi.org/10.1111/cgf.12197 (cit. on pp. 20, 55).

[FSW09] R. Fraedrich, J. Schneider, R. Westermann. "Exploring the Millennium Run - Scalable Rendering of Large-Scale Cosmological Datasets." In: *IEEE Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1251–1258. ISSN: 1077-2626. DOI: 10.1109/TVCG.2009.142 (cit. on pp. 19, 25, 57).

[GGRE13] P. Gralka, S. Grottel, G. Reina, T. Ertl. "Application-specific compression of large MD data preserving physical characteristics." In: *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. Oct. 2013, pp. 85–93. DOI: 10.1109/LDAV.2013.6675162 (cit. on p. 20).

[GHLM05] N. K. Govindaraju, M. Henson, M. C. Lin, D. Manocha. “Interactive Visibility Ordering and Transparency Computations Among Geometric Primitives in Complex Environments.” In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D ’05. Washington, District of Columbia: ACM, 2005, pp. 49–56. ISBN: 1-59593-013-2. DOI: 10.1145/1053427.1053435. URL: http://doi.acm.org/10.1145/1053427.1053435 (cit. on p. 69).

[GIK+07] C. P. Gribble, T. Ize, A. Kensler, I. Wald, S. G. Parker. “A Coherent Grid Traversal Approach to Visualizing Particle-Based Simulation Data.” In: *IEEE Transactions on Visualization and Computer Graphics* 13.4 (July 2007), pp. 758–768. ISSN: 1077-2626. DOI: 10.1109/TVCG.2007.1059. URL: http://dx.doi.org/10.1109/TVCG.2007.1059 (cit. on pp. 20, 54).

[GK15] T. Golla, R. Klein. “Real-time point cloud compression.” In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 5087–5092. DOI: 10.1109/IROS.2015.7354093 (cit. on p. 57).

[GKM+15] S. Grottel, M. Krone, C. Müller, G. Reina, T. Ertl. “MegaMol - A Prototyping Framework for Particle-Based Visualization.” In: *IEEE Transactions on Visualization and Computer Graphics* 21.2 (Feb. 2015), pp. 201–214. ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2350479. URL: http://dx.doi.org/10.1109/TVCG.2014.2350479 (cit. on pp. 20, 33, 88).

[GKM93] N. Greene, M. Kass, G. Miller. “Hierarchical Z-buffer Visibility.” In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’93. Anaheim, CA: ACM, 1993, pp. 231–238. ISBN: 0-89791-601-8. DOI: 10.1145/166117.166147. URL: http://doi.acm.org/10.1145/166117.166147 (cit. on p. 67).

[GKSE12] S. Grottel, M. Krone, K. Scharnowski, T. Ertl. “Object-space ambient occlusion for molecular dynamics.” In: *2012 IEEE Pacific Visualization Symposium*. Feb. 2012, pp. 209–216. DOI: 10.1109/PacificVis.2012.6183593 (cit. on p. 20).

[GLM04] N. K. Govindaraju, M. C. Lin, D. Manocha. “Vis-Sort: Fast Visibility Ordering of 3-D Geometric Primitives.” In: 2004 (cit. on p. 69).

[GM04] E. Gobbetti, F. Marton. “Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models.” In: *Computers & Graphics* 28.6 (2004), pp. 815–826. ISSN: 0097-8493. DOI: http://dx.doi.org/10.1016/j.cag.2004.08.010. URL: http://www.sciencedirect.com/science/article/pii/S0097849304001499 (cit. on pp. 19, 57).

[GNL+15] D. Guo, J. Nie, M. Liang, Y. Wang, Y. Wang, Z. Hu. "View-dependent level-of-detail abstraction for interactive atomistic visualization of biological structures." In: *Computers & Graphics* 52 (Nov. 2015), pp. 62–71. ISSN: 0097-8493. DOI: http://dx.doi.org/10.1016/j.cag.2015.06.008. URL: http://www.sciencedirect.com/science/article/pii/S009784931500093X (cit. on pp. 20, 57).

[GP07] M. Gross, H. Pfister. *Point-Based Graphics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 0123706041, 9780080548821 (cit. on p. 25).

[GRDE10] S. Grottel, G. Reina, C. Dachsbacher, T. Ertl. "Coherent Culling and Shading for Large Molecular Dynamics Visualization." In: *Proceedings of the 12th Eurographics / IEEE - VGTC Conference on Visualization*. EuroVis'10. Bordeaux, France: The Eurographs Association & John Wiley & Sons, Ltd., 2010, pp. 953–962. DOI: 10.1111/j.1467-8659.2009.01698.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2009.01698.x (cit. on pp. 20, 59, 67, 115).

[GSGP06] C. P. Gribble, A. J. Stephens, J. E. Guilkey, S. G. Parker. "Visualizing particle-based simulation datasets on the desktop." In: *Workshop on Combining Visualization and Interaction to Facilitate Scientific Exploration and Discovery*. 2006, pp. 111–118 (cit. on p. 20).

[Gum03] S. Gumhold. "Splatting Illuminated Ellipsoids with Depth Correction." In: *Proceedings of International Workshop on Vision, Modeling, and Visualization*. Nov. 2003, pp. 245–252. URL: http://www.inf.tu-dresden.de/content/institutes/smt/cg/publications/paper/ellipsoid.pdf (cit. on p. 27).

[GZPG10] P. Goswami, Y. Zhang, R. Pajarola, E. Gobbetti. "High Quality Interactive Rendering of Massive Point Models Using Multi-way kd-Trees." In: *2010 18th Pacific Conference on Computer Graphics and Applications*. Sept. 2010, pp. 93–100. DOI: 10.1109/PacificGraphics.2010.20 (cit. on p. 19).

[HE03] M. Hopf, T. Ertl. "Hierarchical Splatting of Scattered Data." In: *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*. VIS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 57–. ISBN: 0-7695-2030-8. DOI: 10.1109/VISUAL.2003.1250404. URL: http://dx.doi.org/10.1109/VISUAL.2003.1250404 (cit. on pp. 19, 25, 57).

[HH11] M. Hapala, V. Havran. "Review: Kd-tree Traversal Algorithms for Ray Tracing." In: *Computer Graphics Forum* 30.1 (2011), pp. 199–213. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2010.01844.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2010.01844.x (cit. on p. 47).

[HM12] L. Hrabcak, A. Masserann. “Asynchronous Buffer Transfers.” In: *OpenGL Insights*. Ed. by P. Cozzi, C. Riccio. http://www.openglinsights.com/. CRC Press, July 2012, First Page-Last Page. ISBN: 978-1439893760 (cit. on pp. 38, 42).

[HOK16] H. Hochstetter, J. Orthmann, A. Kolb. “Adaptive Sampling for On-the-fly Ray Casting of Particle-based Fluids.” In: *Proceedings of High Performance Graphics*. HPG ’16. Dublin, Ireland: Eurographics Association, 2016, pp. 129–138. ISBN: 978-3-03868-008-6. DOI: 10.2312/hpg.20161199. URL: http://dx.doi.org/10.2312/hpg.20161199 (cit. on p. 20).

[KBE09] M. Krone, K. Bidmon, T. Ertl. “Interactive Visualization of Molecular Surface Dynamics.” In: *IEEE Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1391–1398. ISSN: 1077-2626. DOI: 10.1109/TVCG.2009.157 (cit. on p. 20).

[KBR+12] J. Kammerl, N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz, E. Steinbach. “Real-time compression of point cloud streams.” In: *2012 IEEE International Conference on Robotics and Automation*. May 2012, pp. 778–785. DOI: 10.1109/ICRA.2012.6224647 (cit. on p. 57).

[KKL+15] B. Kozlikova, M. Krone, N. Lindow, M. Falk, M. Baaden, D. Baum, I. Viola, J. Parulek, H.-C. Hege. “Visualization of Biomolecular Structures: State of the Art.” In: *Eurographics Conference on Visualization (EuroVis) - STARs*. Ed. by R. Borgo, F. Ganovelli, I. Viola. The Eurographics Association, 2015. DOI: 10.2312/eurovisstar.20151112 (cit. on p. 20).

[KWN+13] A. Knoll, I. Wald, P. A. Navrátil, M. E. Papka, K. P. Gaither. “Ray Tracing and Volume Rendering Large Molecular Data on Multi-core and Many-core Architectures.” In: *Proceedings of the 8th International Workshop on Ultrascale Visualization*. UltraVis ’13. Denver, Colorado: ACM, 2013, 5:1–5:8. ISBN: 978-1-4503-2500-4. DOI: 10.1145/2535571.2535594. URL: http://doi.acm.org/10.1145/2535571.2535594 (cit. on p. 20).

[LBH12] N. Lindow, D. Baum, H.-C. Hege. “Interactive Rendering of Materials and Biological Structures on Atomic and Nanoscopic Scale.” In: *Computer Graphics Forum* (2012). ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2012.03128.x (cit. on pp. 20, 54, 55).

[Lpa14] N. Lukac, D. pelic, B. alik. “Hybrid Visualization of Sparse Point-Based Data Using GPGPU.” In: *2014 Second International Symposium on Computing and Networking*. Dec. 2014, pp. 178–184. DOI: 10.1109/CANDAR.2014.76 (cit. on pp. 19, 25).

[LPC+00] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, D. Fulk. "The Digital Michelangelo Project: 3D Scanning of Large Statues." In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 131–144. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344849. URL: http://dx.doi.org/10.1145/344779.344849 (cit. on p. 18).

[LR71] B. Lee, F. Richards. "The interpretation of protein structures: Estimation of static accessibility." In: *Journal of Molecular Biology* 55.3 (1971), 379–IN4. ISSN: 0022-2836. DOI: http://dx.doi.org/10.1016/0022-2836(71)90324-X. URL: http://www.sciencedirect.com/science/article/pii/002228367190324X (cit. on p. 25).

[Mar17] K. A. Mark Segal. *The OpenGL Graphics System: A Specification (Version 4.5 (Core Profile) - June 29, 2017)*. [Online; accessed 9-July-2017]. The Khronos Group Inc. 2017. URL: https://khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf (cit. on pp. 35, 64).

[Mil68] R. B. Miller. "Response Time in Man-computer Conversational Transactions." In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, part I). 100 milliseconds 100ms ms. San Francisco, California: ACM, 1968, pp. 267–277. DOI: 10.1145/1476589.1476628. URL: http://doi.acm.org/10.1145/1476589.1476628 (cit. on p. 97).

[MKB+15] F. Mwalongo, M. Krone, M. Becher, G. Reina, T. Ertl. "Remote Visualization of Dynamic Molecular Data Using WebGL." In: *Proceedings of the 20th International Conference on 3D Web Technology*. Web3D '15. Heraklion, Crete, Greece: ACM, 2015, pp. 115–122. ISBN: 978-1-4503-3647-5. DOI: 10.1145/2775292.2775307. URL: http://doi.acm.org/10.1145/2775292.2775307 (cit. on p. 20).

[Mye85] B. A. Myers. "The Importance of Percent-done Progress Indicators for Computer-human Interfaces." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '85. 100 milliseconds 100ms ms. San Francisco, California, USA: ACM, 1985, pp. 11–17. ISBN: 0-89791-149-0. DOI: 10.1145/317456.317459. URL: http://doi.acm.org/10.1145/317456.317459 (cit. on p. 97).

[PGA11] R. Pintus, E. Gobbetti, M. Agus. "Real-time Rendering of Massive Unstructured Raw Point Clouds Using Screen-space Operators." In: *Proceedings of the 12th International Conference on Virtual Reality, Archaeology and Cultural Heritage*. VAST'11. Prato, Italy: Eurographics Association, 2011,

pp. 105–112. ISBN: 978-3-905674-34-7. DOI: 10.2312/VAST/VAST11/105-112. URL: http://dx.doi.org/10.2312/VAST/VAST11/105-112 (cit. on p. 19).

[PGX+16] A. Preston, R. Ghods, J. Xie, F. Sauer, N. Leaf, K. L. Ma, E. Rangel, E. Kovacs, K. Heitmann, S. Habib. "An integrated visualization system for interactive analysis of large, heterogeneous cosmology data." In: *2016 IEEE Pacific Visualization Symposium (PacificVis)*. Apr. 2016, pp. 48–55. DOI: 10.1109/PACIFICVIS.2016.7465250 (cit. on p. 19).

[PJR+14] J. Parulek, D. Jönsson, T. Ropinski, S. Bruckner, A. Ynnerman, I. Viola. "Continuous Levels-of-Detail and Visual Abstraction for Seamless Molecular Visualization." In: *Computer Graphics Forum* 33.6 (2014), pp. 276–287. ISSN: 1467-8659. DOI: 10.1111/cgf.12349. URL: http://dx.doi.org/10.1111/cgf.12349 (cit. on p. 20).

[PZBG00] H. Pfister, M. Zwicker, J. van Baar, M. Gross. "Surfels: Surface Elements As Rendering Primitives." In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 335–342. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344936. URL: http://dx.doi.org/10.1145/344779.344936 (cit. on p. 18).

[RCSW14] F. Reichl, M. G. Chajdas, J. Schneider, R. Westermann. "Interactive Rendering of Giga-particle Fluid Simulations." In: *Proceedings of High Performance Graphics*. HPG '14. Lyon, France: Eurographics Association, 2014, pp. 105–116. URL: http://dl.acm.org/citation.cfm?id=2980009.2980021 (cit. on pp. 20, 54, 55, 59).

[RD10] R. Richter, J. Döllner. "Out-of-core Real-time Visualization of Massive 3D Point Clouds." In: *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. AFRIGRAPH '10. Franschhoek, South Africa: ACM, 2010, pp. 121–128. ISBN: 978-1-4503-0118-3. DOI: 10.1145/1811158.1811178. URL: http://doi.acm.org/10.1145/1811158.1811178 (cit. on p. 19).

[RE05] G. Reina, T. Ertl. "Hardware-accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization." In: *Proceedings of the Seventh Joint Eurographics / IEEE VGTC Conference on Visualization*. EUROVIS'05. Leeds, United Kingdom: Eurographics Association, 2005, pp. 177–182. ISBN: 3-905673-19-3. DOI: 10.2312/VisSym/EuroVis05/177-182. URL: http://dx.doi.org/10.2312/VisSym/EuroVis05/177-182 (cit. on p. 20).

[RHI+15] S. Rizzi, M. Hereld, J. Insley, M. E. Papka, T. Uram, V. Vishwanath. "Large-scale Parallel Visualization of Particle-based Simulations Using Point Sprites and Level-of-detail." In: *Proceedings of the 15th Eurographics Symposium on Parallel Graphics and Visualization*. PGV '15. Cagliari, Sardinia, Italy: Eurographics Association, 2015, pp. 1–10. ISBN: 978-3-905674-81-1. DOI: 10.2312/pgv.20151149. URL: http://dx.doi.org/10.2312/pgv.20151149 (cit. on p. 19).

[Ric16] C. Riccio. *OpenGL Mathematics Library (GLM)*. [Online; accessed 9-July-2017]. 2016. URL: http://glm.g-truc.net (cit. on p. 64).

[RK95] J. M. Rehg, T. Kanade. "Model-based tracking of self-occluding articulated objects." In: *Proceedings of IEEE International Conference on Computer Vision*. June 1995, pp. 612–617. DOI: 10.1109/ICCV.1995.466882 (cit. on p. 69).

[RL00] S. Rusinkiewicz, M. Levoy. "QSplat: A Multiresolution Point Rendering System for Large Meshes." In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 343–352. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344940. URL: http://dx.doi.org/10.1145/344779.344940 (cit. on pp. 18, 25).

[RSH01] E. Reinhard, P. Shirley, C. Hansen. "Parallel Point Reprojection." In: *Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics*. PVG '01. San Diego, California: IEEE Press, 2001, pp. 29–35. ISBN: 0-7803-7223-9. URL: http://dl.acm.org/citation.cfm?id=502125.502130 (cit. on p. 80).

[RTW13] F. Reichl, M. Treib, R. Westermann. "Visualization of big SPH simulations via compressed octree grids." In: *2013 IEEE International Conference on Big Data*. Institute of Electrical and Electronics Engineers (IEEE), Oct. 2013, pp. 71–78. DOI: 10.1109/BigData.2013.6691717. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6691717 (cit. on pp. 25, 57, 59).

[SBD+15] A. Scherzinger, T. Brix, D. Drees, A. Volker, K. Radkov, N. Santalidis, A. Fieguth, K. Hinrichs. "Visualize the universe: Interactive exploration of cosmological dark matter simulation data." In: *2015 IEEE Scientific Visualization Conference (SciVis)*. Oct. 2015, pp. 115–135. DOI: 10.1109/SciVis.2015.7429500 (cit. on p. 19).

[SGG15] J. Staib, S. Grottel, S. Gumhold. "Visualization of Particle-based Data with Transparency and Ambient Occlusion." In: *Computer Graphics Forum* 34.3 (2015), pp. 151–160. ISSN: 1467-8659. DOI: 10.1111/cgf.12627. URL: http://dx.doi.org/10.1111/cgf.12627 (cit. on pp. 20, 115).

[SMK+16] K. Schatz, C. Müller, M. Krone, J. Schneider, G. Reina, T. Ertl. "Interactive visual exploration of a trillion particles." In: *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*. Oct. 2016, pp. 56–64. DOI: 10.1109/LDAV.2016.7874310 (cit. on pp. 19, 25).

[SMW98] C. T. Silva, J. S. B. Mitchell, P. L. Williams. "An exact interactive time visibility ordering algorithm for polyhedral cell complexes." In: *IEEE Symposium on Volume Visualization (Cat. No.989EX300)*. Oct. 1998, pp. 87–94. DOI: 10.1109/SVV.1998.729589 (cit. on p. 69).

[SPL04] M. Sainz, R. Pajarola, R. Lario. "Points Reloaded: Point-based Rendering Revisited." In: *Proceedings of the First Eurographics Conference on Point-Based Graphics*. SPBG'04. Switzerland: Eurographics Association, 2004, pp. 121–128. ISBN: 3-905673-09-6. DOI: 10.2312/SPBG/SPBG04/121-128. URL: http://dx.doi.org/10.2312/SPBG/SPBG04/121-128 (cit. on p. 18).

[ST90] T. Saito, T. Takahashi. "Comprehensible Rendering of 3-D Shapes." In: *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '90. Dallas, TX, USA: ACM, 1990, pp. 197–206. ISBN: 0-89791-344-2. DOI: 10.1145/97879.97901. URL: http://doi.acm.org/10.1145/97879.97901 (cit. on p. 85).

[SVGR16] R. Skånberg, P. P. Vázquez, V. Guallar, T. Ropinski. "Real-Time Molecular Visualization Supporting Diffuse Interreflections and Ambient Occlusion." In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 718–727. ISSN: 1077-2626. DOI: 10.1109/TVCG.2015.2467293 (cit. on p. 20).

[TS91] S. J. Teller, C. H. Séquin. "Visibility Preprocessing for Interactive Walkthroughs." In: *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '91. New York, NY, USA: ACM, 1991, pp. 61–70. ISBN: 0-89791-436-8. DOI: 10.1145/122718.122725. URL: http://doi.acm.org/10.1145/122718.122725 (cit. on p. 67).

[TUH+14] S. Tanaka, M. Uemura, K. Hasegawa, T. Kitagawa, T. Yoshida, A. Sugiyama, H. T. Tanaka, A. Okamoto, N. Sakamoto, K. Koyamada. "Application of Stochastic Point-Based Rendering to Transparent Visualization of Large-Scale Laser-Scanned Data of 3D Cultural Assets." In: *2014 IEEE Pacific Visualization Symposium*. Mar. 2014, pp. 267–271. DOI: 10.1109/PacificVis.2014.25 (cit. on p. 19).

[Unk11] A. Unknown. *View Frustum Culling*. Lighthouse3d.com. 2011. URL: http://www.lighthouse3d.com/tutorials/view-frustum-culling/radar-approach-testing-spheres/ (cit. on p. 66).

[VHB14] M. Vinkler, V. Havran, J. Bittner. “Bounding Volume Hierarchies Versus Kd-trees on Contemporary Many-core Architectures.” In: *Proceedings of the 30th Spring Conference on Computer Graphics*. SCCG ’14. Smolenice, Slovakia: ACM, 2014, pp. 29–36. ISBN: 978-1-4503-3070-1. DOI: 10.1145/2643188.2643196. URL: http://doi.acm.org/10.1145/2643188.2643196 (cit. on p. 47).

[WBMS05] A. Williams, S. Barrus, R. K. Morley, P. Shirley. “An Efficient and Robust Ray-box Intersection Algorithm.” In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH ’05. Los Angeles, California: ACM, 2005. DOI: 10.1145/1198555.1198748. URL: http://doi.acm.org/10.1145/1198555.1198748 (cit. on p. 50).

[WDG02] B. Walter, G. Drettakis, D. P. Greenberg. “Enhancing and Optimizing the Render Cache.” In: *Proceedings of the 13th Eurographics Workshop on Rendering*. EGRW ’02. Pisa, Italy: Eurographics Association, 2002, pp. 37–42. ISBN: 1-58113-534-3. URL: http://dl.acm.org/citation.cfm?id=581896.581901 (cit. on p. 80).

[WDP99] B. Walter, G. Drettakis, S. Parker. “Interactive Rendering Using the Render Cache.” In: *Proceedings of the 10th Eurographics Conference on Rendering*. EGWR’99. Granada, Spain: Eurographics Association, 1999, pp. 19–30. ISBN: 3-211-83382-X. DOI: 10.2312/EGWR/EGWR99/019-030. URL: http://dx.doi.org/10.2312/EGWR/EGWR99/019-030 (cit. on p. 80).

[WE97] R. Westermann, T. Ertl. “The VSBUFFER: visibility ordering of unstructured volume primitives by polygon drawing.” In: *Visualization ’97., Proceedings*. Oct. 1997, pp. 35–42. DOI: 10.1109/VISUAL.1997.663853 (cit. on p. 69).

[WH06] I. Wald, V. Havran. “On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N).” In: *2006 IEEE Symposium on Interactive Ray Tracing*. Sept. 2006, pp. 61–69. DOI: 10.1109/RT.2006.280216 (cit. on p. 47).

[WIK+06] I. Wald, T. Ize, A. Kensler, A. Knoll, S. G. Parker. “Ray Tracing Animated Scenes Using Coherent Grid Traversal.” In: *ACM SIGGRAPH 2006 Papers*. SIGGRAPH ’06. Boston, Massachusetts: ACM, 2006, pp. 485–493. ISBN: 1-59593-364-6. DOI: 10.1145/1179352.1141913. URL: http://doi.acm.org/10.1145/1179352.1141913 (cit. on p. 54).

[Wik15a] O. Wiki. *Buffer Object Streaming — OpenGL Wiki,* [Online; accessed 8-July-2017]. 2015. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Buffer_Object_Streaming&oldid=12456 (cit. on p. 38).

[Wik15b] O. Wiki. *Early Fragment Test — OpenGL Wiki,* [Online; accessed 29-June-2017]. 2015. URL: %5Curl%7Bhttp://www.khronos.org/opengl/wiki_opengl/index.php?title=Early_Fragment_Test&oldid=12704%7D (cit. on p. 35).

[Wik16] O. Wiki. *Fragment — OpenGL Wiki,* [Online; accessed 11-July-2017]. 2016. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Fragment&oldid=13722 (cit. on p. 92).

[Wik17a] O. Wiki. *Buffer Object — OpenGL Wiki,* [Online; accessed 11-July-2017]. 2017. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Buffer_Object&oldid=13857 (cit. on p. 31).

[Wik17b] O. Wiki. *Primitive Assembly — OpenGL Wiki,* [Online; accessed 11-July-2017]. 2017. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Primitive_Assembly&oldid=13850 (cit. on p. 85).

[Wil92] P. L. Williams. "Visibility-ordering Meshed Polyhedra." In: *ACM Trans. Graph.* 11.2 (Apr. 1992), pp. 103–126. ISSN: 0730-0301. DOI: 10.1145/130826.130899. URL: http://doi.acm.org/10.1145/130826.130899 (cit. on p. 69).

[WJA+17] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, P. Navratil. "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization." In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 931–940. ISSN: 1077-2626. DOI: 10.1109/TVCG.2016.2599041 (cit. on p. 20).

[WKJ+15] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, M. E. Papka. "CPU ray tracing large particle data with balanced P-k-d trees." In: *2015 IEEE Scientific Visualization Conference (SciVis)*. Oct. 2015, pp. 57–64. DOI: 10.1109/SciVis.2015.7429492 (cit. on pp. 20, 22, 26, 47, 49, 50, 113).

[WLWD03] C. Woolley, D. Luebke, B. Watson, A. Dayal. "Interruptible Rendering." In: *Proceedings of the 2003 Symposium on Interactive 3D Graphics*. I3D '03. Monterey, California: ACM, 2003, pp. 143–151. ISBN: 1-58113-645-5. DOI: 10.1145/641480.641509. URL: http://doi.acm.org/10.1145/641480.641509 (cit. on p. 79).

[WS06] M. Wimmer, C. Scheiblauer. "Instant Points: Fast Rendering of Unprocessed Point Clouds." In: *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*. SPBG'06. Boston, Massachusetts: Eurographics Association, 2006, pp. 129–137. ISBN: 3-905673-32-0. DOI: 10.2312/SPBG/SPBG06/129-136. URL: http://dx.doi.org/10.2312/SPBG/SPBG06/129-136 (cit. on pp. 19, 25).

[WSWG13] D. Wodniok, A. Schulz, S. Widmer, M. Goesele. “Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays.” In: *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization*. EGPGV ’13. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2013, pp. 57–64. ISBN: 978-3-905674-45-3. URL: http://tubiblio.ulb.tu-darmstadt.de/82620/ (cit. on p. 47).

[WWB+14] I. Wald, S. Woop, C. Benthin, G. S. Johnson, M. Ernst. “Embree: A Kernel Framework for Efficient CPU Ray Tracing.” In: *ACM Trans. Graph.* 33.4 (July 2014), 143:1–143:8. ISSN: 0730-0301. DOI: 10.1145/2601097.2601199. URL: http://doi.acm.org/10.1145/2601097.2601199 (cit. on p. 115).

[ZD15] T. Zirr, C. Dachsbacher. “Memory-efficient On-the-fly Voxelization of Particle Data.” In: *Proceedings of the 15th Eurographics Symposium on Parallel Graphics and Visualization*. PGV ’15. Cagliari, Sardinia, Italy: Eurographics Association, 2015, pp. 11–18. ISBN: 978-3-905674-81-1. DOI: 10.2312/pgv.20151150. URL: http://dx.doi.org/10.2312/pgv.20151150 (cit. on pp. 20, 59).

[ZPBG01] M. Zwicker, H. Pfister, J. van Baar, M. Gross. “Surface Splatting.” In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’01. New York, NY, USA: ACM, 2001, pp. 371–378. ISBN: 1-58113-374-X. DOI: 10.1145/383259.383300. URL: http://doi.acm.org/10.1145/383259.383300 (cit. on p. 18).

All links were last followed on July 19, 2017.

**Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature