Point-based Visualization of Molecular Dynamics Data Sets

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Sebastian Grottel

aus Sindelfingen

Hauptberichter:	Prof. Dr. Thomas Ertl
Mitberichter:	Prof. Dr. Stefan Gumhold

Tag der mündlichen Prüfung: 29. März 2012

Visualisierungsinstitut der Universität Stuttgart

2012

Contents

Contents ... 3
Acronyms ... 5
Abstract ... 7
1 Introduction ... 9
1.1 Particle-based Data ... 9
1.1.1 Definition of particle-based Data ... 12
1.1.2 Sources of particle-based Data ... 14
1.2 Point-based Visualization ... 21
1.3 Software Design for Visualization Applications ... 23
2 Optimized Particle-based Rendering ... 27
2.1 Glyph Ray Casting ... 29
2.2 Data Transfer ... 37
2.2.1 Graphics Hardware Bus ... 37
2.2.2 Loading from Secondary Storage ... 46
2.3 Occlusion Culling ... 50
2.4 Deferred Shading & Ambient Occlusion ... 57
2.4.1 Image-Space Normal Estimation ... 58
2.4.2 Ambient Occlusion for Particle Data Sets ... 59
2.5 Complex and Composed Glyphs ... 74
2.5.1 Glyphs Composed from Quadric Surfaces ... 76
2.5.2 Polyhedral Glyphs ... 78
3 Derived Feature Visualization ... 87
3.1 Extraction of spatial Structures ... 87
3.1.1 Ellipsoid Representation for Particle Clusters ... 88
3.1.2 Particle Clusters as Image-Space Metaballs ... 90
3.1.3 Isosurface in Particle Density Volume ... 99
3.1.4 Molecular Surface ... 102
3.1.5 Linear and Planar Crystal Defects in Atom Lattices ... 107
3.2 Visualization of Temporal Features ... 114
3.2.1 Loose Capacity-Constrained Path Lines ... 115
3.2.2 Principal Particle Path Lines ... 124
3.2.3 Dynamic Interaction between Molecule Clusters ... 129
4 MegaMol ... 137
4.1 Visualization System Engineering ... 137
4.2 VIS***lib*** – a Reusable Source-Code Library ... 144
4.3 MegaMol – Framework Design ... 149
5 Conclusion ... 158
Zusammenfassung – German Abstract ... 167
List of Figures ... 171
List of Tables ... 181
Bibliography ... 183

Acronyms

API .... Application Programming Interface
BCC .... Body-Centred Cubic
CCVD .... Capacity-Contrained Voronoi Diagram
CPU .... Central Processing Unit cf. GPU
CUDA .... Compute Unified Device Architecture
DEM .... Discrete Element Method
DMA .... Direct Memory Access (data transfer)
FBO .... Frame-Buffer Object
FCC .... Face-Centred Cubic
fps .... Frames per Second
GLSL .... OpenGL Shading Language
GPGPU .... General Purpose GPU
GPU .... Graphics Processing Unit
HCP .... Hexagonal-Close Packed
HWOQ .... hardware-supported occlusion query
HZB .... Hierarchical maximum-Depth Buffer
LCCVD .... Loose Capacity-Constrained Voronoi Diagram
MD .... Molecular Dynamics
MSS .... Molecular Skin Surface
OSAO .... Object-Space Ambient Occlusion
PBC .... Periodic boundary conditions
PCI-E .... PCI Express (Peripheral Component Interconnect Express)
SAS .... Solvent Accessible Surface (Molecular Surface)
SDK .... Software Development Kit
SE .... Software Engineering
SES .... Solvent Excluded Surface (Molecular Surface)
SFB .... Sonderforschungsbereich, German Collaborative Research Center
SPH .... Smooth Particle Hydrodynamics
SSAO .... Screen-Space Ambient Occlusion
VA .... OpenGL Vertex Array
VBO .... Vertex-Buffer Object
VdW .... Van-der-Waals (Molecular Surface)
μ-CT .... Micro Computer Tomography (aka. Industrial CT Scanning)

Abstract

The analysis and especially the exploration of large data sets from simulations often benefit from visualization. If it is not possible to calculate some well-known characteristic values, or if it is known that spatial features like distributions may play an important role, directly looking at a visual representation of the data becomes a first and important step in the analysis process. This is especially true for scientific work where simulations with novel methods or novel problem scenarios produce data with potentially unknown features and effects. Particle-based simulation methods, like molecular dynamics, smooth particle hydrodynamics, or the discrete element method are prominent examples as these simulation techniques themselves, as well as the scenarios they are applied to, are in the focus of current research of the corresponding application fields, e.g. physics, thermodynamics, biochemistry, material science and engineering. The sizes of the simulation scenarios, and as a consequence the sizes of the resulting data sets, steadily increased during the last years to close the gap between length scales accessible through experiments and length scales reachable by simulations. This is not only because the increased compute power of individual machines or the availability of relatively cheap compute clusters to be used on-site, but also because of significant improvements of the simulation algorithms themselves.

Most of the available visualization tools are insufficiently optimized to cope with the data set sizes they have to face nowadays, as they often require the whole data set to be stored in main memory or exclusively use mesh-based rendering methods. For rendering particle-based data sets point-based ray casting has become the state-of-the-art approach and large data sets are typically addressed by different data-streaming and out-of-core techniques. This thesis investigates the performance bottlenecks of secondary storage and the data transfer between main memory and graphics hardware, as well as the impact of different compression techniques using spatial data structures and quantisation of coordinates, to optimize the foundation for such streaming techniques. Point-based ray casting is presented for different kinds of graphical primitives, like spheres, cylinders, compound glyphs, and polyhedral crystallites, extending the applicability of this rendering approach. To optimize the rendering performance the required calculations are reduced to the necessary minimum employing advanced culling techniques, allowing for interactive visualization of data sets with hundreds of millions of particles.

A second and probably more important aspect when visualising large particle data sets remains: creating meaningful and useful visualizations. Even a system capable of rendering millions of particles will usually generate images that are prone to aliasing, visual clutter, and other effects hindering good perception and thus the understanding of the presented data. This thesis presents two approaches of advanced shading and lighting to remedy this issue. An image-space method to estimate normal vectors for the structure implicitly formed by the particles addresses the aliasing problem, while the perception of the global structure and depth

of the data is enhanced by a specially adapted ambient occlusion technique approximating global illumination.

For more efficient visualizations, relevant structures need to be derived from the original particle data. Such consolidated visualizations provide a better overview of the structure of the data sets. However, as they reduce the visual information, such representations must be created with care to be sure not to omit important data or introduce misleading artefacts from the applied methods. This thesis presents several examples, highlighting two aspects: spatial structures and representatives for the dynamics of the data. The examples of the first group range over molecular clusters, i.e. droplets in the context of thermodynamics nucleation simulations, dislocations and stacking faults from material science, and generic surface representations, similar to molecule surface descriptions, known from biochemistry. The dynamics of data are given by examples of the interaction of molecule clusters, the clustering of path lines of water-protein interactions and the tracking of expelled material in laser ablation simulations.

To reliably avoid problems, like introducing artefacts or misleading presentations, with derived representations it is important to always involve experts from the corresponding application domain, because only those are able to judge usefulness and correctness of a visualization compared to the original data. Such close collaborations are most fruitful if the application domain expert can also be the actual user of the visualization tools, e.g. allowing to experiment with parameter settings. This, however, requires the visualization software to be usable for people other than computer scientists. Especially in academia, scientific software is often written only as fragile prototype programs, capable of showing the functionality of an algorithm as proof-of-concept. Most of the time there is no simple, usable user interface or thorough error handling. Applying best practices of software engineering provides a way to remedy this problem, creating high-quality software, ready to be used by end users, with only slightly increased development efforts. This thesis presents a process, as well as some guidelines to introduce just the right amount of software engineering into the development of scientific visualization software. MegaMol, a system focussed on point-based visualization for particle data sets, serves as example for the effectiveness of this process.

All the methods in this thesis together form a consistent solution for effective and efficient visualization of large data sets with hundreds of millions of particles, by addressing the problems of rendering and data loading performance, perception, feature extraction and tracking. A suitable development process to create the required, complex visualization software in academic environments is also discussed. Although all the presented approaches focus on point-based visualization of molecular dynamics data sets, their generic nature allows for their application in other scenarios as well.

1 Introduction

Computer-based simulations are nowadays widely employed in different fields of science to complement costly, dangerous, or physically impossible experiments. With the on-going increase of available compute power at low prices, the problems tackled with these simulations also increase. Nowadays data sets from simulations in everyday work in the fields of physics or thermodynamics consist of millions of atoms and thousands of time steps. These data sets thus easily reach several gigabytes, even terabytes, in size on a daily work basis; sizes that were only considered by national laboratories a few years ago. While these large data sets are necessary to model scenarios and materials at a scale relevant to study the phenomena the research aimes for, the overwhelming amount of data continuously poses a challenge for the task of analysis. If the process to derive the sought answers from the simulation data is well known, and ideally automatable, the analysis process is merely a question of calling the right programs or scripts. However, in natural sciences these processes are often not clear or, even worse, it is unknown which features of the data will yield the answers and explanations the researchers look for. This is not only true for simulations in the field of molecular dynamics (MD), but for all sciences. This problem is addressed by excessively analysing and observing the data. Observing raw data of the nowadays reached sizes is often impossible, or at least inefficient and most of the time also ineffective. At this point the importance of good, suitable visualisation gains more and more importance.

The purpose of visualization is to gain insight as [CMS99] states. The visualization handbook by Hansen and Johnson [JH04] explains visualization as “enabling role in our ability to comprehend [...] large and complex data” and that it “conveys insight into [...] diverse applications”. This applies to the mentioned visual analysis of MD data sets in the context of physics, thermodynamics, biochemistry and other fields of application. A good, i.e. effective and efficient, visualization aids a human user in understanding the data, by changing the user’s knowledge [vW05]. There are many different efforts to specify a formal definition for visualization, which all have in common that visualization is to help researchers gain information and understanding from abstract or symbolic data through visual representations. The focus of this thesis is on visualization for particle data sets from MD simulations, but the presented approaches and algorithms are, despite their optimizations, also applicable to other data as well. This is especially true for the content of chapters 3 and 4.

1.1 Particle-based Data

As visualization is meant to map from raw data, e.g. from MD simulations, to visual representations to support human observers in deriving knowledge and insight, the structure of this data is of high importance. Data sets form MD simulations, or simi-

lar sources, have a unique structure compared to other data presented in visualization as they consist of attributed entities (molecules or atoms) freely floating in a physical space, thus different from classical visualisation data, which is usually based on 2D or 3D grids of different topology with attributes attached to the grid vertices and interpolated in-between. Data sets with unconnected or uncorrelated entities are frequently called *point-based data* or *mesh-less data*. Data sets of these types often describe discrete samples of a continuous signal. One example is surfaces, e.g. scanned from real objects with laser scanners and represented by a set of points in 3D space. Such point set surfaces are usually visualised by point-based rendering of simple surface splats (circular or elliptical splats [BK03]). Point-based rendering of large data sets has been quite an active area of research in the last few years. Many algorithms deal with the point-based rendering of extremely large geometry, or point set surfaces, first using the CPU [BWK02], [ZPvBG01], then using programmable graphics hardware [RPZ02], [PSG04], [TGN+06]. An example is shown in the left image of Figure 1. The rendering quality of such surfaces has been steadily improved over the years to yield high surface quality while maintaining interactive rendering performance (e.g. see [BHZK05] and [OHS06] for more information). A good overview over the field of point-based graphics is given in the correspondingly named book [GP07].

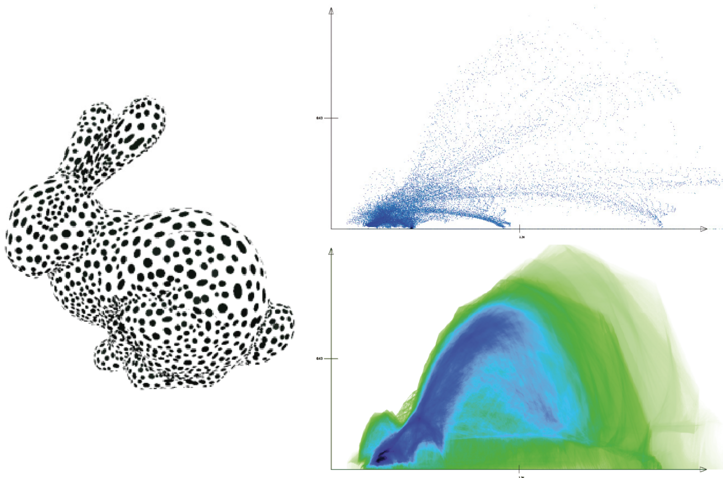

Figure 1: Point-based surface rendering of a point-based data representation of the Stanford Bunny at coarse resolution (left) [GP07]; classical scatterplot (right, top) of the “blunt-fin” data set and continous scatterplot (right, bottom) of the same data [BW08]

In uncorrelated point-based data sets, usually called *scattered data*, often the individual point does not hold relevant information by its own, but only the continuous structure of the whole point set conveys the information, like local differences in density distributions. This information is visible, e.g., in scatter plots, which can therefore also be evaluated and represented in continuous fashion, as presented in [BW08]. The right image of Figure 1 shows examples from these types of data sets.

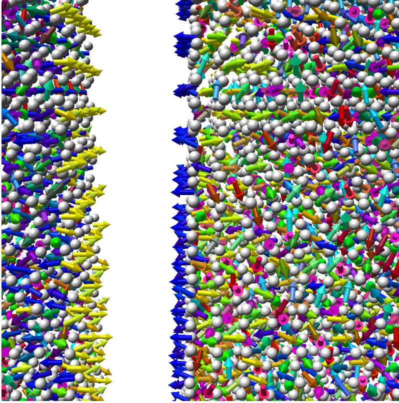

Figure 2: Detail of a visualization of orientations of polarized atoms in cracked solid alumina (unpublished collaboration between D.3, B.1, and B.2 of SFB 716[1]). While the polarizable atoms are orientated randomly in the bulk, the dipole momenta align at the surface of the crack.

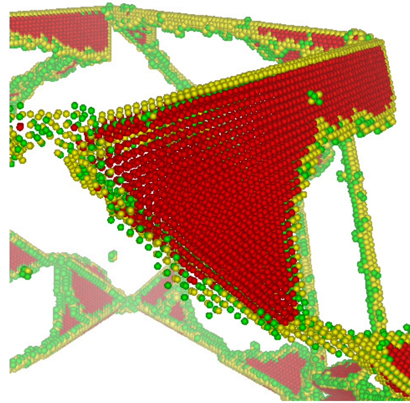

Figure 3: A planar stacking fault surrounded by linear dislocations within solid crystalline metal (NiAl), all formed by individual atoms [GDCE09]; the structures are classified and coloured based on the common-neighbour analysis [HA87], identifying FCC structures, BCC structures, and atoms with otherwise arranged 12 nearest neighbours (cf. Chap. 3.1.5).

MD data sets are different and must be treated differently, as the information of individual atoms can be of high importance for the analysis process, as well as the discrete description of a continuum, usually a bulk e.g. a gas, liquid or solid, without explicit geometric properties. For example, on one hand, a charge of a single atom within a molecule can define polarity and thus the behaviour of the molecule as a whole, making this information of that atom relevant. On the other hand, the simulated material as a whole is formed by all atoms and conveys additional information, like atom densities or the shapes of molecules or macromole-

[1]SFB 716 [DFG]: D.3 - Visualization of Systems with Large Numbers of Particles; B.1 - Molecular Dynamics of Large Systems with Long Range Interactions; B.2 - Molecular dynamics simulation of the fracture of metal/ceramics interfaces

cules. Thus, data sets from MD exhibit a duality of interests, as the individual elements contain relevant information (Figure 2) as well as the continuous structures formed by all particles (Figure 3). Because of this special characteristic the term *particle-based data* is used in this thesis to clearly address such data sets.

1.1.1 Definition of particle-based Data

Data sets are defined differently for different visualizations. There are several possible understandings of the same data structures. For the discussion in this chapter representations have been chosen to clarify the special properties of particle-based data sets compared to other data sets common in scientific visualization. Especially the term of the dimensionality of the data is ambiguous across different applications. In information visualization, a data entity is usually given values for multiple attributes and a whole data set is often a list of such entities. Here the number of attributes m, i.e. the number of dependent variables, define the dimensionality of the data:

$$\mathbb{N} \rightarrow A^m,$$

which is a mapping of some sort of entity index to the set of attribute values. Time-dependence is introduced in this kind of data sets by either adding a point-in-time value to the attributes A^m or by extending the index, i.e. independent variables, correspondingly:

$$\mathbb{N} \times \mathbb{R} \rightarrow A^m. \tag{1}$$

In this case the dimensionality of the data set is not changed by introducing time dependency. The data set is still referred to as m-dimensional data. This is fundamentally different from, e.g., common volume or vector data sets, which are prominent cases for scientific visualization. There, the data entities holding the attribute data, i.e. voxels, are explicitly or implicitly related to a spatial position, and the dimensionality of this positional information is usually understood as the dimensionality of the data:

$$\mathbb{R}^n \rightarrow A^m,$$

with n usually being 2 or 3, resulting in the naming of 2D data or 3D data. Time-dependency can be added to this kind of data in a similar way as before: by adding the time to the independent variables:

$$\mathbb{R}^n \times \mathbb{R} \rightarrow A^m.$$

The term of 4D data stems from this approach, e.g. for time-dependent volume data ($n = 3$). The actually implemented data storage may differ from this theoretical definition, especially for irregular data grids, in which flexible voxel placement as well as explicit connectivity needs to be stored.

Considering these definitions, particle data, even though in the context of scientific visualization, is more closely related to the list-like data of information visualization: each particle is a data entity with several attached attributes, and the data set as a whole is a list of these particles. The values for the attributes of one particle are addressed by a particle index and a specific point in time (cf. Equation (1)). However, unlike data sets from information visualization, particle data sets are usually related to a physical space, similar to classical volume data sets. The coordinates of the particle are thus part of its attached attributes:

$$A^m = \mathbb{R}^n \times A^{m-n}.$$

The number n of these attributes related to the physical space is usually understood as the dimensionality of the data set. The remaining $m - n$ attributes store values like temperature or potential energy. As time-dependence is introduced by an independent variable, but the dimensionality of the data is based on a specific subset of dependent variables, it does not make sense to speak of 4D particle data when meaning time-dependent 3D data. Thus, terms like *trajectory*, *time-dependent* or *time-varying particle data sets* are better suited to describe the nature of the data and to avoid any misunderstandings. Again, a concrete implementation of the data storage might differ from these definitions. Especially, many particle attributes in A^{m-n} may be constant over time or even constant for all particles. These values can thus be stored globally outside the list of particles. For mixtures of particles of different types, these global values can be stored as look-up table, based on a particle type identifier in A^{m-n}. Then, atoms of different types should also be stored in separate lists, to optimize for the visualization (not only because of memory concerns but also to optimize data transfer; cf. chapter 2).

Equation (1) uses a continuous time dimension. Obviously, the time dimension has to be discretised, similar to the spatial dimensions. In MD simulation the time is discretised into *time steps*. At the beginning and the end of each time step the state of all particles is fixed and can be visualized as static points in time. Based on terminology from computer graphics in this thesis the term *time frame* will be used to denote such a static configuration of a time-dependent data set, similar to an image frame of an animation. In the simulation, the transition from one time frame to the next one, which is a single time step, is realised by numeric integration. To minimize the errors of this integration the time steps of the run simulations are usually of the order of one femtosecond (10^{-15}s). The processes and effects to be studied usually happen on the time scale of nanoseconds (10^{-9}s) at best. Thus, useful simulations and the resulting data sets are bound to have at least 10^6 time steps.

Often, one is not only interested in the distribution and density inside a material or in geometric structures implicitly formed, but also in properties of the individual molecules the simulation creates, like their temperature or charge. Usually only the one or two values which are most interesting for the analysis task at hand are then stored in the attribute vector, and later mapped to visual attributes, like the colour of the particle. A value worth noting explicitly is the speed vector of a particle, which often is related the temperature of the atom in a thermodynamic context. On first sight, this seems like redundant information, as this information can be reconstructed from the positional information from different time steps (e.g. forward differences). However, this would only be true if the data to be visualised would contain all time steps calculated by the simulation. Visualizing all time steps is not reasonable as the changes and effects to be observed usually happen on much larger time scales. Visualization therefore often uses only every 100-th or 1000-th time step of the simulated data. Reconstructing speed vectors from such reduced data would filter out high-frequent movement (e.g. Brownian motion), leaving only the principal movements (e.g. laminar flow) and would thus underestimate the speed vector significantly.

Some particle types require an orientation to be stored. Molecules are often modelled as structures of multiple mass centres without inner degrees of freedom (not necessarily using the exact atoms forming the molecule in a chemical context). Even if the shape is approximately spherical, the molecules can have non-uniform distributions of changes, making them polar and making the orientation of the polarity relevant. The common practice is to store the orientation using a normalized quaternion.

Those features are the most commonly found attributes in MD data sets. Of course, specific applications require further attributes. Some examples are presented in the chapters 2 and 3.

1.1.2 Sources of particle-based Data

Particle-based simulations are probably the most important and the most widely-used source for such data sets. While it is not necessary to fully understand all details of these simulations to create useful visual representations, an understanding of the fundamental methods and their limitations is crucial to judge the effectiveness of the resulting visualization. Especially, visualization of features derived from the original data might introduce artefacts. The visualization researcher has to be able to differentiate between artefacts which were already present in the original data, e.g. due to the method of the simulation, or if those were introduced by the calculations for the visualization. Therefore, this chapter gives a short overview over common sources of particle data sets from a visualization scientist's point of view.

The method of MD, as prominent example of particle-based simulation techniques, was introduced by McCammon et al. [MGK77]. In MD simulations a number of entities, usually atoms or mass elements representing molecules or parts of molecules, are modelled as attributed points. These entities interact with each other using pair potentials: between each pair of entities a usually attracting force is calculated, primarily from the distances and masses of the points in space, but also considering other attributes, like charges, if required by the specific application. All these forces are summed up for each entity resulting in the overall force applied. Based on these forces the motion of the entities is calculated by integration of the classical Newtonian equations of motion. Details on the simulation technique can be found e.g. in [GKZ07]. The result of such simulations are several time frames, each storing the list of entities with the values of their attributes at this specific point in time (e.g. position, kinetic energy, etc.), which can be also interpreted as a list of particles with their attributes changing over time.

This principle is used in a wide range of different simulation methods, like n-body simulation, quantum mechanical simulation, smooth particle hydrodynamics (SPH), and discrete element method (DEM). Each method adopts the principle to its specific needs, e.g. the DEM, since it is concerning macroscopic particles, like stones, usually requires explicit handling of contacts between the particles and evaluates friction between them. Despite their differences all methods have in common that, at least in theory, all particles might interact with all other particles resulting in a runtime-complexity of $\mathcal{O}(n^2)$. Since many phenomena to be studied with these simulations require very large numbers of particles to generate statistically relevant results, this is not feasible. However, some assumptions are possible which can remedy this issue. The probably most prominent one is the use of a cut-off distance: usually the strength of the interaction forces between two entities decreases fast as the distance increases. Therefore, a cut-off distance can be chosen where the force between two particles at this distance or a larger distance can be regarded as zero. This way only a very limited neighbourhood around each particle needs to be evaluated. Another possibility is to evaluate the forces with a more coarse-grained method. For example, in SPH, introduced by Gingold and Monaghan [GM77], the force on each particle can be calculated from the gradient of the density of the particles' distribution. These can be calculated in a continuous field representation, allowing for reduction of the calculations from billions of particle-pairs to only calculation of thousands of values on the mesh of the field. These methods are thus called *Particle Mesh*. Apart from these intrinsic forces, often an additional global force can be applied. This force allows for stimulating a laminar flow or allows simulating material behaviour under external force, like performing tensile testing [CSSS06]. Several works in MD discuss the evaluation of material properties under such external forces. Also, see Bulatov et al. [BC06] for a good introductory book on this subject. The principle of using a continuous force field to move the particles is also applied to visualize and analyse properties for such vector fields themselves using the particles only as visual entities. This is often used in context

of flow visualization. However, even introducing this field would go beyond the scope of this thesis. As an example, the article of Krüger et al. [KKKW05] is a good introduction into this field.

Other simulation techniques and specialized implementations exist. Kipfer et al. [KSW04] presented a GPU-based system for particle simulations, which was, however, primarily intended for rendering and animation and not scientific simulation. A similar approach was presented by Kolb et al. [KLR04] in the same year and was extended to simulate coupled particles, i.e. particles reacting with each other and not only reacting to an underlying velocity field [KC05], allowing a Lagrangian particle simulation of fluids. Van Meel et al. [vMAF+07] used the modern general purpose GPU (GPGPU) API CUDA for a fully functional molecular dynamics simulation on graphics hardware. A smoothed particle hydrodynamics simulation on the GPU was presented by Zhang et al. [ZSP08] to simulate and render liquids for computer graphics.

Simulations must take care of how to handle the boundary of the simulated area. There are two common approaches: open boundaries and periodic boundary conditions (PBC). There are other methods to explicitly create a boundary, e.g. defining particles (atoms or molecules) which interact normally with all other particles, but are fixed in their position. This way enclosing geometry, like a nanoscale channel can be modelled. However, such methods are rather specialized constructs for particular applications. With open boundaries the simulation space extends theoretically to infinity, but is not filled with any simulated entities. This models the subject of the simulation, e.g. a block of solid or liquid material, to be surrounded by vacuum. PBC *repeat* the simulated area infinitely often in all directions. While this allows modelling an more or less homogeneously filled infinite volume, e.g. a vapour, the fact that the simulation area is connected to itself can create subtle problems. While problems like searching all neighbours of one particle within a given radius when using PBCs is only an implementation issue, larger phenomena in *one simulation area* might affect the *neighbouring simulation areas*, thus the one simulation area itself. For example, consider an event occurring at a specific location creating a shockwave moving through the simulated particles, e.g. an energy pulse hitting some molecules. This shockwave will re-enter the simulation area when leaving it. These *echoing* or *repetition* effects must be kept in mind when analysing the data. A further similar effect is shown in Figure 4. Consider a simulation of vapour which is cooled down to become a liquid. A bulk of atoms in vacuum will end up as spherical drop, and the result is likely to be the same when using periodic boundaries. However, if the simulation area is not equally sized in all directions (e.g. the size in z-direction is significantly shorter than the sizes of the other two directions) an infinitely long cylinder of liquid connected to itself across the boundaries becomes a possible, stable final state for the simulation.

Despite the problems and computational complexity PBCs come with several benefits. Often the effects from echoing or self-connection are negligible compared to the problems of surrounding vacuum or the computational effort of explic-

it boundaries (which also often expose reflection effects similar to the described echoing). Charged particles in electrostatic simulations can create long-ranged interactions between particles. To capture these effects a very large cut-off radius would be required, which then would result in poor performance. Advanced simulation methods have been developed to remedy this issue: e.g. the Ewald Summation [Ewa21] separates the electrostatic interactions into a short-range and a long-range component. While the short-range component can be calculated using a feasible cut-off radius, the long-range component can be efficiently evaluated numerically in Fourier space, which however requires the simulation to use PBCs in all directions.

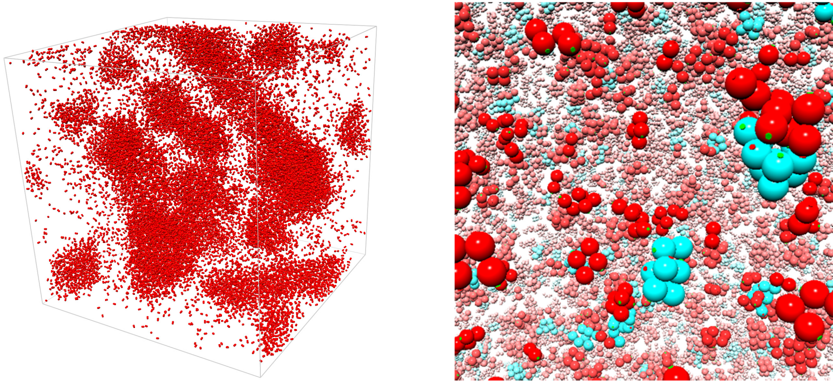

Figure 4: Thermodynamics MD simulation of super-saturated vapor (CO_2 (left) and R-152a (right)) forming molecule clusters (coloured cyan in the right image) as predecessors of droplets during the process of nucleation; usually these pre-droplets would be spherical because of surface tension. Because of the extreme preasure and the PBCs the molecule clusters in the left image instead connect to each other across different instances of the simulation area as a rather stable end state.

An example for typical MD simulation from the field of thermodynamics is nucleation, i.e. the state change of a material e.g. from a vapour into the liquid phase. This process is found in many physical phenomena, e.g. the formation of atmospheric clouds or the processes inside steam turbines. A detailed knowledge of the dynamics of condensation processes can help to optimise energy efficiency here and to avoid problems with droplets of macroscopic size [For04]. Starting with data sets at a homogeneous state point in the vapour phase, either the pressure is increased or the temperature is decreased such that a metastable state point in the two-phase region is reached. The liquid phase emerges through spontaneous density fluctuations in the vapour which lead to the formation of nuclei (molecular clusters), the predecessors of liquid droplets. These clusters reach macroscopic size

through aggregation of individual molecules or coalescence with other clusters. Figure 4 show two nucleation data sets. In the right image, the molecular clusters are coloured in cyan to be better distinguishable from the surrounding vapour. This nucleation process is still a research topic for thermodynamics researchers and is being investigated at the molecular level with the help of large-scale simulation runs. The key properties of these processes are the nucleation rate and the critical cluster size. The critical cluster size designates clusters with a certain number of molecules that have the same probability of growth as of decay; smaller clusters will more likely evaporate and bigger clusters will more likely continue to grow. The nucleation rate quantifies the number of emerging clusters beyond the critical size per volume and time. There is a classical nucleation theory [VW26] describing this condensation process. However, it does not predict nucleation rates with satisfactory precision. In many cases the predicted values deviate from experimental findings by several orders of magnitude [FRK+98]. This situation gets even worse when working with mixtures. MD simulation promises to remedy this issue as to allow detailed insight into the processes on the molecular level. This is also true for other phase transitions, like melting and freezing [HA87]. Adequate models for the intermolecular interactions containing all relevant thermodynamic properties of real fluids have become available [VSH01], [SVH06], with which the predicted nucleation rates are much more accurate. However, as mentioned earlier, as the nucleation is a process based on probabilities, rather large systems with several million molecules must be simulated over a long period of time to reach cluster sizes up to thousands of molecules and to detect low nucleation rates. Only then the simulation can be compared to and verified against real-world conditions in most technical processes.

A further source of particle-based data sets is stochastic construction. Based on values like particle sizes, distribution, and overall density a particle data set can be reconstructed by a point processes. Often, the initial atom placement setup for MD simulations is carried out in such a way. Given the simulation area and the number of atoms to be placed, e.g. derived from a targeted density, the initial atom positions can be computed, either from random numbers or following atom lattice definitions like FCC or BCC for mono-crystal materials. This task gets more complicated when thinking about complex materials, like alloys. There, e.g. two materials, both with different values for lattice cell lengths, are combined in a single lattice, introducing lattice defects (e.g. dislocations). While this is a simple case on the micro-scale, it gets more complicated on the meso- or macro-scale, where the materials are built up by grains or crystallites, forming shapes which cannot meaningfully be reduced to spherical particles. Such materials, e.g. porous media [Hil96] like sandstone, often exhibit complex microstructure, even enclosing void space. Real material probes are scanned, e.g. using μ-CT, and afterwards analysed to derive the stochastic material parameters required for synthesis. Figure 5 compares a scan of Fontainebleau sandstone with the stochastic reconstruction. Such stochastic reconstruction models for porous media have been investigated e.g. in [MTH00].

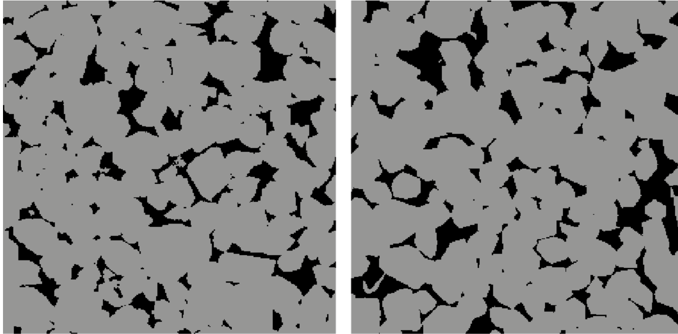

Figure 5: Comparison of 2D sections (2.25 mm × 2.25 mm) experimental data (μ-CT; left) with reconstructed model (right) of Fontainebleau sandstone. Grains are shown in grey, while space in-between is shown in black. The reconstructed model is stochastically very similar to the experimental data. The degree of stochastic similarity was measured and documented quantitatively using numerous geometric observables described in detail in [LBFH10].

Since such material is built up from grains or crystallites, these can be described in a particle-based fashion: each particle stores its position, size, orientation and shape. The material modelling often starts with models using spherical grains, because of the simplicity of handling interactions, to resemble density and distribution within the original material. However, for realistic results non-spherical particles are required to recreate the often thin interconnections between enclosed void spaces. One possible way of doing this is modelling the material using multiple subgrain spheres, an approach often chosen when working with DEM [JBPE99]. However, this approach obviously does not scale to system sizes of 10^8 grains, which is the size required to achieve results comparable to laboratory experiments. Nowadays the material is therefore constructed from polyhedral particles [LBFH10], reducing the amount of particles to be simulated, while at the same time, however, complicating the process of evaluating contact points and forces. Because of this, visually inspecting the material model is of high importance for the analysis process. Visualization of such stochastic models is important for identifying and modelling regions of interest in multi-scale porous media, such as micro-porous regions with sub-micron-sized pores or microcrystalline regions with nanometre crystallites. An exact representation of the particle shapes is required as they define the shape and amount of the surface of the media in cavities and tunnel networks exposed to surrounding media like gas or liquid (e.g. consider thick-film gas sensors [MC95]). Figure 6 shows an example 3D rendering of a small reconstructed sandstone model, created through point-based visualization of polyhedral particles (100 particle templates with 18 faces each).

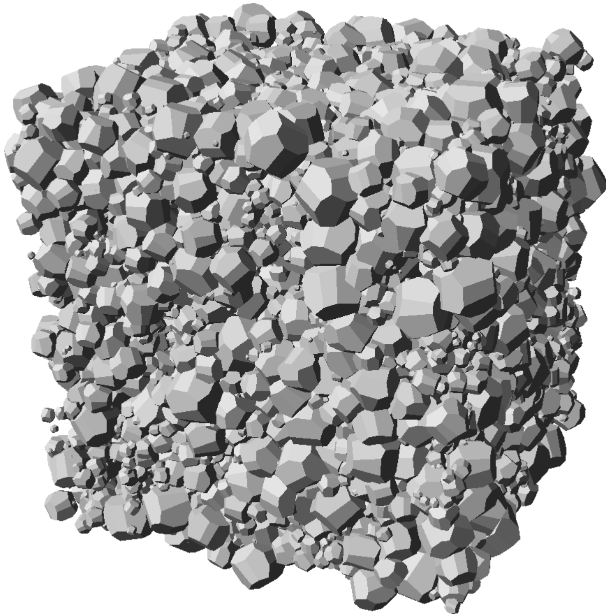

Figure 6: A small porous media sample modelled with 10 000 particles using 100 crystallite templates. Each particle is a scaled and rotated instance of one of these templates.

A second example of particle data being reconstructed from measured real-world objects is X-ray crystallography, which has long been used to analyse proteins and other macromolecules in context of biochemistry. The protein data base [BWF+00] is a huge archive of protein models obtained by this method. The same measuring technique has also been used to analyse water solvent at protein surfaces, since crystal structures determined at high resolution provide a detailed picture of protein hydration [Nak04]. For example, protein-solvent interactions have been studied for a microbial antibiotic resistance protein using long-time scale MD simulations. Comparing different scans of the same protein or of slight variants is the key aspect in this task. Analysing several data sets revealed positions repeatedly occupied by water molecules [BWT06]. Already available protein-centric algorithms are able to identify so-called water bridges [San04], which correspond with these positions. The particle-based visualization approach is valid for the macromolecules, i.e. proteins, as well as the solution molecules surrounding them. The most apparent distinction is that the protein has inner degrees of freedom between the atoms forming it, while the solvent molecules are usually modelled as being

static without inner degrees of freedom, similar to the more coarse-grained models applied in thermodynamics.

1.2 Point-based Visualization

Visualization is an essential part of the analysis workflow when working with particle-based data sets. Only in very rare cases the data can be evaluated automatically and summarized to only a few values that can be efficiently interpreted by the researcher. An example might be the analysis of material properties e.g. in thermodynamics, where a single meaningful value could be calculated from the data by well-known techniques. However, especially if the data contains phenomena not completely understood or if the researcher simply does not know for sure what she or he is looking for, visualising and exploring the data becomes the best choice. This is not novel and visualizations are already used, e.g. in form of diagrams and histograms. An example is the use of speed diagrams of a laminar flow in a nano-sized channel as shown in Figure 7 (cf. [HVBH09]). However, problems arise as soon as the spatial information from the data set is required and cannot be broken down to only one or two values. Then the need for three-dimensional visualization emerges, which instantly yields the need for interactive visualization, as 3D representations are almost always bound to suffer from occlusion or poorly chosen viewing angles. This is why there already exists a great variety of visualization tools for particle data sets, which greatly differ in focus, performance, and features.

The visualization of molecular dynamics data has been approached with various different visualization techniques over the years. The fundamental visualization method which suits this kind of data sets best is glyph-based visualization. Glyphs, e.g. known from information visualization [CE97], are visual representatives of individual entities depicting multiple attributes at once, but also convey information about the collective of whole data sets. However, discussing glyph-based visualization in a generic fashion is too far off the focus of this thesis. Instead, only glyph-based visualization in the context of particle data sets is presented. Some of the most well-known tools for glyph-based MD data visualization are Chimera [PGH+04], PyMOL [DeL02], and VMD [HDS96]. Generic visualization packages, such as AVS [AVS] or Amira [Ami] also provide special modules for molecular visualization. These tools focus on analysis features and offer many functions corresponding to the specific application domains. However, their rendering performance often does not allow for interactivity when rendering very large data sets, since most of them use polygon-based representations without sophisticated level-of-detail techniques. Another problem for large time-dependent data sets is that most of these tools require the whole data set being loaded into main memory. Carefully optimized viewers, such as TexMol [BDST04], BioBrowser [HOF05], or the tool presented in [Max04], provide higher performance by harnessing GPU capabilities and thus achieve better interactivity. However, these optimized rendering tools, in

general, lack many analysis features of the generic tools like VMD, as they focus on the visualization of the particle data.

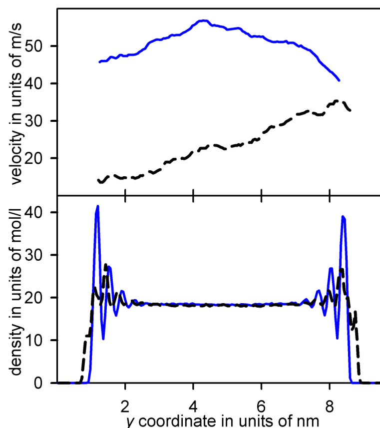

Figure 7: Example diagram for typical visualization of data from particle simulation with one spatial axis: a velocity profile (top) and density profile (bottom) for Poiseuille (solid lines) and Couette (dotted lines) flow of liquid methane at a temperature 166.3 K within a carbon nano-channel with width of 8 nm and a characteristic flow velocity of 50 m/s [HVBH09].

There are further examples and optimized approaches for specific tasks: e.g. utilizing a mixed environment of a graphical workstation together with a compute cluster for scene optimization [SNK+03] or multi-resolution rendering [NKV99] allowing for interactive visualization of large data sets, but these are limited in their application. Hopf and Ertl presented a GPU-optimized level-of-detail method for point-based astronomy data sets [HE03]. This approach is based on textured splats representing the particles. Similar approaches exist for opaque glyphs – e.g. spheres – as well [BDST04], but these approaches typically achieve lower visual quality as the limited resolution of the normal and depth textures and quantization effects cause visual artefacts. Gumhold has shown how to ray cast implicit quadric glyph surfaces on the GPU to be able to render several thousands of particles [Gum03]. Klein and Ertl presented a similar approach for rendering ellipsoids [KE04], which was optimized by utilizing the point primitive of the OpenGL graphics API. Reina and Ertl [RE05] showed how this approach could be used to visualize compound glyphs using the example of a *dipole glyph* being ray cast with a single shader pass. Current programs like Qutemol [TCM06] not only offer good rendering performance, but also enhance the perceptibility employing advanced shading techniques like ambient occlusion and edge cueing. GPU ray casting has also been applied for special visualizations like molecule surfaces [KBE09] or molecule clus-

ters [MGE07]. The point-based ray casting method allows for comprehensive optimizations enabling the rendering of very large data sets with up to hundreds of millions of particles on a standard desktop workstation [GRDE10]. As this topic is the focus of this thesis, detailed information can be found in chapter 2.

1.3 Software Design for Visualization Applications

An essential part of visualization research is the development of new algorithms, the optimization of known concepts, and the application of these findings to new problems and tasks provided by application scenarios, e.g. from different fields of natural sciences. This, almost always, includes optimized implementations of the algorithms, at least on the level of prototype programs as proof of concept and for performance evaluation. Common practice, as consequence from the limited time available to create these implementations, is to write these prototypes following a simple *on-need-basis* strategy: only functionality which is required of this very project will be implemented and almost no planning ahead will take place. However, basic functions, like the rendering main loop, parsing of command line options, or loading of data files, is always the same for the prototype software written. Creating each visualization prototype from scratch again is obviously not reasonable. Thus, these functions are often simply copied from an older prototype into a new implementation. Bugs fixed in the previous program migrate into the new prototype, but due to the omitted planning for future needs, the original implementations often require alteration or extension resulting in degraded stability or maintainability of the source code, lowering the benefit for future reuse. To remedy this problem these functions need to be developed in a centralized and advanced fashion instead, following basic principles of *software engineering* (SE), allowing for future use and extension in a controllable and maintainable way.

For many people in industry and academia the idea of SE is often rather vague and frequently associated with the allegation of *making easy tasks complicated*. However, the IEEE Computer Society gives a quite clear definition, which is *the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software* [Ei90]. Processes which have proven to be efficient and effective in engineering should be used in software development. These are based on some basic principles: the work is meant to solve a problem; evaluation is based on practical success and costs; awareness of the quality of the final product is required, as well as thinking in standards and modules. When developing software in academia, not least in fields of computer science like visualization, almost all of these principles are constantly ignored: problems solved here have oftentimes no direct connection to real-world applications; practical success is replaced by acceptance for publication; quality awareness and coding standards do factually not exist; and thinking in modules boils down to thinking in C++ classes at best. To some degree, this is un-

derstandable as programs are seen as prototypical proof-of-concept implementations of a proposed new method – the publication is the product. Following a software development process model [Jal97] does, indeed, introduce overhead. In addition, many models are too inflexible for the fuzzy requirements definitions in academia, and newer, more agile processes usually rely on an external customer defining and constantly checking the requirements.

So why abandon the usual way of tinkering with software [Bau93]? Prototypes are only meant to get a first impression of a future product. In the research context, programs are mostly used for performance measures and screen shots. Often, they even cannot serve any other purpose. Sometimes, even loading data sets other than the ones used for the publication might fail. However, there are two important reasons for developing software with better quality: 1. often academic work, especially in scientific visualization, happens in close collaboration with non-computer-scientist users from an application domain and 2. most visualization research nowadays is incremental [Dub05] and thus the need to reuse or recycle software increases.

Concerning users from an application domain, visualization software to be used needs a decent level of stability and usability. Making visualization implementations available to these users is highly beneficial when tackling real-world problems as part of the visualization research. In such scenarios the methods and approaches for visualization must be derived from the data available and the goals the application domain scientists try to achieve. Usually neither the domain scientists nor the visualization researchers are experts in both fields, and thus an optimal approach to reach a suitable visualization must be found in close collaboration and discussions. One way to assist this task is to make a visualization prototype available to both parties as experimenting with the visualization will nurture further ideas for improvements and extensions from both sides. Working together on the same visual representations will also help clarifying the scientific vocabulary from both the application domain and computer science.

As for the issues of doing incremental work, implementations of previous methods are always needed, either as basis to continue on or to be evaluated for performance comparisons with new implementations. Here it is obvious that both implementations need the same amount of optimization work put into them to gain relevant results – e.g. comparing a highly optimized GPU implementation against a quickly written, single-threaded CPU version is neither fair nor useful, but common practice. Especially as the focus for visualization research shifts back from just making a (GPU-accelerated) rendering possible to solving problems from an application domain employing visualization, the need for high-quality software increases, whereas current programs are often so poorly written that no domain scientist can use them on their own.

Thus, the difficulty lies in finding a process flexible enough for the research context and minimizing the SE overhead on the one hand, while ensuring sufficient software quality on the other hand, hence allowing the programs to be used in real

applications and allow reuse of as much source code as possible. The core concept is quite simple: creating and reusing of *reusable software components*. Compared to quickly-written visualization prototypes, creating these components involves significantly more time, even when applying modern, agile processes of SE, but only once. Through each reuse of the components this additional initial effort pays off, and due to the clean development the stability and maintainability of these codes is almost not affected by further alterations or extensions, which will only happen sparsely, because of the initial design planning. After a decent stock of re-usable components has been developed, visualization prototypes are implemented more quickly and reach a higher software quality. Especially the software framework of these prototypes, responsible for fundamental tasks can highly benefit from this approach, as there will be no need to implement it, but it can be simply composed from the available components. Thus, the researcher can fully focus on the implementation of the new algorithms and visualization methods.

Obviously, as each researcher in a group will face the same problems concerning software the fundamental code components are likely to be the same for each one. It thus makes sense to collect these components, when they are cleanly written with the aim of reuse, in such a way that the whole group can utilize them. This does not mean that a software system should be created for the group and that each researcher has to implement his or her algorithms and visualizations as part of this software. Instead, collecting these code components in form of a loose group of interoperable classes, or similar, allows each researcher to only pick and use the components required for a specific task. Based on this rather generic collection of source code modules for a whole research group, each individual researcher can build up a smaller and more specialized collection of modules and classes especially tailored for his or her research area. These modules can also be shared among multiple persons working in similar directions, but they are meant for a smaller audience.

Chapter 4 will present a software framework created based on these principles for visualization prototypes in the field of particle visualization: MegaMol. The process of integrating SE in a research group without introducing too much overhead and hindrances for its introduction will be detailed along this example, with explanations and motivations of design decisions, pitfalls and opportunities.

2 Optimized Particle-based Rendering

Point-based visualization seems like an obvious choice for particle-based data sets. In fact, this approach is an optimal representation of the data, as it is true to the original simulation results. It thus forms a ground truth for any visual analysis process. The basic idea of point-based visualization is that each element in the data set, in the case of data sets from MD simulations atoms or rigid molecules, is depicted by a compact graphical element – a glyph – representing the main attributes of the data element. There are several different approaches for implementing glyph-based visualizations.

Following classical computer graphics, the polygon-based approach would be to create a triangle mesh for the glyph's geometry of each particle. As computer graphics hardware is especially designed to handle large numbers of triangles this approach seems reasonable. However, many glyph representations are composed of curved, smooth surfaces. A prominent example is the sphere, which is probably the most often used glyph, especially in the field of molecular and atomistic visualization. Triangulating a sphere, or even only the visible hemi-sphere, requires a significant amount of triangles. Obviously, the resolution of the triangulation can be adapted to the image-space size of the sphere, i.e. the viewer's distance. Spheres with a size of only a single pixel can easily be approximated by a single triangle (as smallest entity in the classical rendering approach to produce a single fragment). However, as the size of the sphere in image-space increases, the number of triangles must also be increased, e.g. by employing a geometry shader or a tessellation shader. Both introduce much variance into the rendering pipeline, because each glyph will require a different number of output triangles, corresponding to its image-space size. This hampers the GPU to reach the possible peak performance, which is only achievable through rather rigid GPU-side data multiplication, e.g. by using instancing [Ins]. It is thus obvious that this approach is not able to scale to the large numbers of particles state-of-the-art MD data sets contain.

A second approach for glyph rendering is to use billboards, i.e. image-plane-aligned quads showing an image of the glyph, usually loaded from a texture. This approach is simple for rotation-symmetric glyphs like spheres, if effects like perspective distortion are ignored. Such an approach was presented in [BDST04]. In principle, four vertices are generated for each particle defining the image-space quad of the correct size. OpenGL offers an alternative: the *point* primitive, which expands to an image-space square in the rasterization stage. The point-sprite extension for OpenGL even offered the automatic generation of texture coordinates, allowing to simply applying a texture holding the image of the glyph without the need for any shader program. This approach was extended further by applying simple shader programs, e.g. by outputting corrected depth information, allowing the glyphs to correctly intersect each other utilizing the standard depth test, or by using a normal texture instead of a colour texture to calculate correct lighting of the glyph based on the scene light setup.

Today's graphics hardware tries to eliminate legacy functionality, focusing on only the most important elements, which, due to the high level of programmability, can be used for all tasks the dismissed functionalities were used for. E.g. DirectX does not have a graphical primitive *point*. Instead a simple geometry shader can be used to generate the image-space geometry (a quad constructed from two triangles) from a single uploaded vertex. This quad can then be used in the same fashion the point-sprites were used.

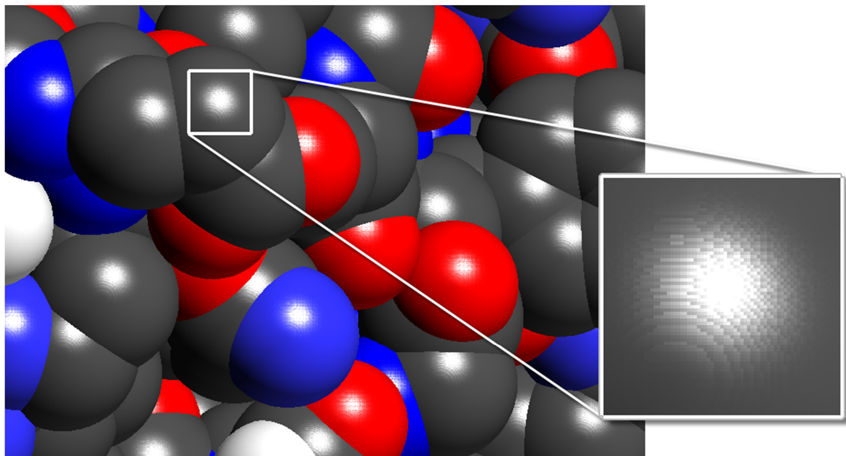

Figure 8: Example of visual artefacts from using a texture-based rendering approach for particle data sets: precision of normal textures are insufficient; Image was rendered by TexMol [BDST04] (image from [Rei08]).

However, these approaches require a texture holding the information of the image of the glyph to generate the final representation. This comes with two drawbacks: first, all texture-based methods always share the problem of a finite maximum resolution of the texture. If the texture is too small, visual artefacts will appear due to precision issues, which e.g. can be seen in Figure 8 (usually linear or nearest-neighbour interpolation is applied, as higher order interpolations would require more data values, resulting in rather expensive texel fetches, as well as a higher calculation effort, as only up to tri-linear interpolation is available as hardware implementations in the graphics cards). The optimal display quality is only reached if the texture holding the glyph's image is exactly stored at the same resolution as the image-space size of the glyph. Current graphics card offer high calculation performance, while the memory access required for the texel fetch has limited potential for optimisation from a graphics applications point-of-view. Thus, the idea is to calculate the information of the glyph's image directly at the required

resolution for each glyph instead of fetching the information from a texture. This leads to the idea of *ray casting*: ray tracing without secondary rays, therefore also known as *local ray tracing*.

2.1 Glyph Ray Casting

The core idea of *ray tracing* is to trace the light energy as it is distributed throughout the scene from the light sources, over the surfaces of the different objects, finally into the eye of the observer, i.e. a camera, or in a reverse fashion from the observer in the specific direction of a single fragment, through the scene, back to the light sources. The required viewing ray through the image pixel is easily calculated from the observer position, viewing direction and camera parameters, e.g. aperture angle. This viewing ray is then intersected with the closest object surface. From this surface hit point, the light is further tracked back, directly to visible light source and through reflexion or refraction rays (secondary rays) into other directions.

Ray casting simplifies this technique by only performing local lighting calculations (usually employing the well-known method of Blinn [Bli77]) at the found surface hit point, thus only requiring local surface information, like reflective colour and the surface normal vector, for the known light sources. Therefore, this approach does not intrinsically capture shadows or reflections. Together with the previously described image-space splatting for glyphs ray casting allows for precise per-fragment evaluation and lighting of the surface of a glyph, thus creating the apparent image for each glyph anew, at the exact resolution required.

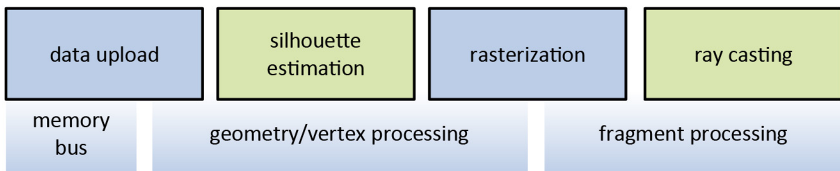

Figure 9: Simplified process of GPU-based ray casting: The blue stages are part of the rendering hardware, while the green stages are operations of the ray casting. First the size of the glyph's silhouette is estimated in vertex processing to span an image-space quad. For the fragments of this quad the ray casting is performed.

The fundamental principles of glyph ray casting were initially presented by Gumhold [Gum03], who described how to efficiently evaluate a viewing ray surface intersection with programmable graphics hardware. The surface of the glyph is described by a quadratic equation, allowing for a general solution of many surfaces of this class, e.g. sphere, ellipsoid, cylinder and cone. Klein and Ertl [KE04] presented a similar approach, focussed on ellipsoids to aid their visualization goals, but utiliz-

ing the point primitive of OpenGL to optimize the required data transfer between GPU and main memory. Figure 9 illustrates the principal steps of this technique. These glyph images are perspective-correct and can intersect each other applying the built-in OpenGL depth test. The fragment shader stage is used to perform the viewing ray surface intersection. In the vertex shader stage the graphical primitive which will generate the fragments for this glyph is fit to an approximation of the final silhouette of the glyph in image space. The works of Toledo and Levy [TL04] and later by Sigg et al. [SWBG06] use the same principles.

The probably simplest glyph of this type, and also the most commonly one used, is the sphere:

$$|\boldsymbol{x} - \boldsymbol{p}| = r$$

with $\boldsymbol{p}$ being the position and r the radius of the sphere. Thus, $\boldsymbol{x}$ describes all points on the surface of the sphere. For simplicity of the required ray casting calculation the sphere can be placed at the origin of the coordinate system, eliminating $\boldsymbol{p}$. The ray casting equation is then given by:

$$|\boldsymbol{c} + \lambda\boldsymbol{v}| = r$$
$$(c_x + \lambda v_x)^2 + \left(c_y + \lambda v_y\right)^2 + (c_z + \lambda v_z)^2 = r^2$$

with $\boldsymbol{c}$ being the position of the camera, $\boldsymbol{v}$ being the normalized viewing vector derived from the current fragment, and λ being the ray casting parameter. Evaluating lambda for a given set of parameters (camera, viewing ray and radius) only results in real values if the viewing ray actually hits the sphere:

$$\begin{aligned} \lambda^2\left(v_x^2 + v_y^2 + v_z^2\right) + 2\lambda\left(c_x v_x + c_y v_y + c_z v_z\right) + c_x^2 + c_y^2 + c_z^2 - r^2 = 0 \\ \lambda^2(\boldsymbol{v} \cdot \boldsymbol{v}) + 2\lambda\,(\boldsymbol{c} \cdot \boldsymbol{v}) + (\boldsymbol{c} \cdot \boldsymbol{c}) - r^2 = 0 \\ \lambda = \frac{-2(\boldsymbol{c} \cdot \boldsymbol{v}) \pm \sqrt{4(\boldsymbol{c} \cdot \boldsymbol{v})^2 - 4(\boldsymbol{v} \cdot \boldsymbol{v})(\boldsymbol{c} \cdot \boldsymbol{c} - r^2)}}{2(\boldsymbol{v} \cdot \boldsymbol{v})} \end{aligned} \quad (2)$$

Note that the evaluation of the dot product is optimal for graphics hardware, as traditionally the graphics processing cores of graphics cards were designed to operate on 4D vector variables (nowadays, GPUs operate on scalar values). Since the spherical glyph is defined by a quadratic surface equation, there are two solutions of λ in Equation (2). These correspond to the front-side and back-side hit of the sphere with the viewing ray. As only the front-side hit is relevant for opaque spheres and λ represents the distance of the intersection point from the camera, only the smaller value is relevant (subtracting the value of the root). A similar equation can be given for all other surfaces described by quadratic functions. However,

some of them have an unlimited extension in at least some direction, e.g. cylinders and must be explicitly cut resulting in branching shader code.

Given the hit point $\boldsymbol{x} = \boldsymbol{c} + \lambda\boldsymbol{v}$ a surface normal can be calculated. For the case of a sphere placed at the origin of the coordinate system it is given by $\boldsymbol{n} = \boldsymbol{x}/r$. The surface normal vector and the hit point position provide enough information, together with arbitrarily chosen surface parameters, to perform local lighting operations, resulting in the final image.

The shown operation is to be performed for each fragment of each glyph. A naïve approach would be to render full-screen image-space quads, either one for each glyph or a single one, iterating over all glyphs in the corresponding shader. Obviously, both approaches are not acceptable, because of the huge number of unnecessary calculations. The solution is to estimate the size of the footprint of the glyph in image space, the tight-fitting bounding rectangle of all fragments which generate rays that will hit the surface of the object, before actually generating any fragments. This estimation of the silhouette of the glyph in image space has to be computed in the vertex shader stage, before the rasterization and thus the fragment generation takes place. If the image-space quad is to be generated by the geometry shader, as mentioned above, the silhouette estimation needs to be evaluated in that stage instead.

For specific shapes for the glyphs, like closed quadratic surfaces, e.g. spheres or cylinders, the image space silhouette can be directly calculated analytically. While the ray casting itself is a backward projection: computing the object surface intersection point based on a fragment, the silhouette must be calculated as forward projection: finding the image-space bounding box, based on the surface parameters. In theory all surface points are projected from object-space into image-space and the bounding rectangle enclosing all points is evaluated. Obviously the goal is to only transfer the four points on the surface which will contribute to the final result, i.e. the points which will have the minimum/maximum x/y-coordinate after projection.

Spheres are simply defined by a position and radius in object space. Finding the four contributing points on the surface of the sphere is analytically possible and has been shown in [Rei08] before. The image-space bounding rectangle corresponds to an asymmetric pyramidal frustum, thus given by four planes with common attributes: all planes go through the camera position and all touch the sphere as tangent planes. Additionally the normal vectors of the planes are reduced in their degrees of freedom as they have additional constraints from the shape of the frustum: for the plane defining the horizontal extent in image-space the y-component of the normal vector needs to be zero. Thus these calculations can be reduced to a problem in 2D: x-z-plane for the horizontal extents and y-z-plane for the vertical extends. Figure 10 depicts the principle of calculating this extent for a sphere at position $\boldsymbol{p}$ with radius r. The solution is given by the two contact points $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ of the sphere and the tangent planes through the camera position $\boldsymbol{c}$. Considering the triangle $\boldsymbol{cpb}_2$ the values of the distances p, q, and h are given by Euclid' theorems:

$$p = \frac{r^2}{|\boldsymbol{p}-\boldsymbol{c}|} \qquad q = |\boldsymbol{p}-\boldsymbol{c}| - p \qquad h = \sqrt{pq}$$

Given $\boldsymbol{v} = \boldsymbol{p} - \boldsymbol{c}/|\boldsymbol{p}-\boldsymbol{c}|$ being the normalized vector from the camera position to the position of the sphere and $\boldsymbol{v}'$ forming a orthonormal basis with $\boldsymbol{v}$ in this 2D plane the requested points $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ can easily be calculated:

$$\boldsymbol{b}_{1,2} = \boldsymbol{p} - p\boldsymbol{v} \pm h\boldsymbol{v}'$$

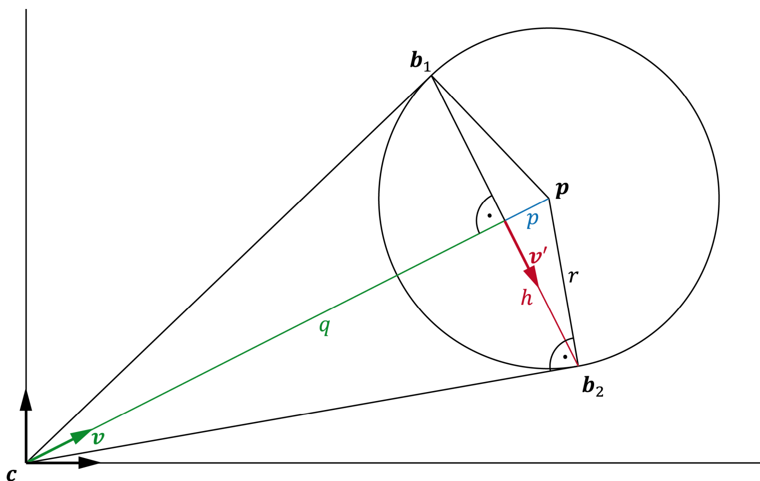

Figure 10: Evaluation of the horizontal image-space extent of a sphere at position $\boldsymbol{p}$ with radius r; The image-space extent as seen from camera $\boldsymbol{c}$ is given by the points $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$.

Note, since we are working in 2D, $\boldsymbol{v}'$ can be created from $\boldsymbol{v}$ by simply swizzling its two coordinates and by negating one of them. This corresponds to multiplication with the 2D rotation matrix for a rotation of 90°. Thus the calculation of the four points in object-space which will span the image-space bounding rectangle is computationally cheap. This approach was further optimized in [Kle08] for ellipsoidal glyphs finding a bounding polygon in image space. For the data sets presented there, which contained rather elongated ellipsoids, bounding hexagons resulted in the best rendering performance, i.e. best trade-off between overhead of image-space footprint estimation and actually saved fragment calculations.

While utilizing tangent planes works for spheres, it is obviously not generalizable for arbitrarily shaped or even compound glyphs. However, the generic idea of

forward-projecting relevant points of the glyph into object-space instead of all visible surface points remains valid. As the silhouette in image-space is only an approximation anyway a second approximation offers a solution: an object-space bounding box is calculated for the glyph at first, and, then the eight corner points of this bounding box are projected into image-space to form the bounding rectangle as glyph approximation. This rough approximation highly over-estimates the image-space foot-print, but it is a valid alternative, when calculating a tight-fitting bounding rectangle would be too demanding in terms of processing load.

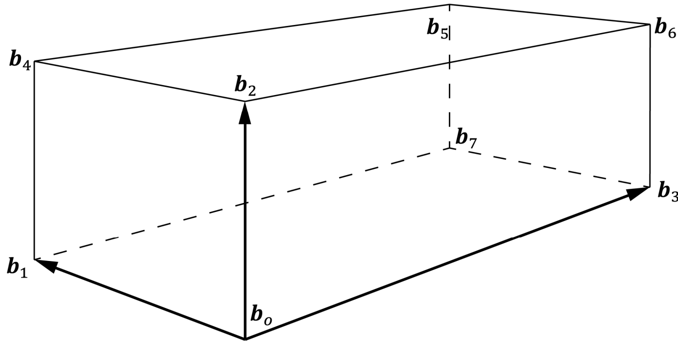

Figure 11: Definition of three main axis vectors of an objects-space bounding box for reconstruction in image space

For this approach, the probably most expensive calculations is the matrix-vector multiplications to project the points of the object-space bounding box into image-space. It is possible to reduce the number of these projections to a minimum of four, by utilise the fact that the only calculation of the projection from object-space into image-space that is non-linear is the perspective division. Thus scalar multiplications and vector additions can be performed any time before this operation. An optimized evaluation of the image-space foot-print therefore projects only four points $\boldsymbol{b}_0$, $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, and $\boldsymbol{b}_3$ of the object-space bounding box, yielding $\boldsymbol{b}'_0$, $\boldsymbol{b}'_1$, $\boldsymbol{b}'_2$, and $\boldsymbol{b}'_3$ (cf. Figure 11). These four points can then be used to build up a coordinate system in image-space before the perspective division, which can be used to calculate the remaining four points: e.g. $\boldsymbol{b}'_5 = \boldsymbol{b}'_2 + (\boldsymbol{b}'_3 - \boldsymbol{b}'_0) + (\boldsymbol{b}'_1 - \boldsymbol{b}'_0)$. The cost for the required calculations is thus of the same order as for the evaluation of the bounding rectangle for spheres.

Using this approach, image-space footprints can be estimated for arbitrarily complex glyphs. Obviously, the resulting image-space silhouette will be over-estimated; especially if an additional approximation by an OpenGL *point* square is used. Therefore, while still feasible for complex or compound glyphs, it is favourable to build specialized solutions for important and common glyphs, whenever possible.

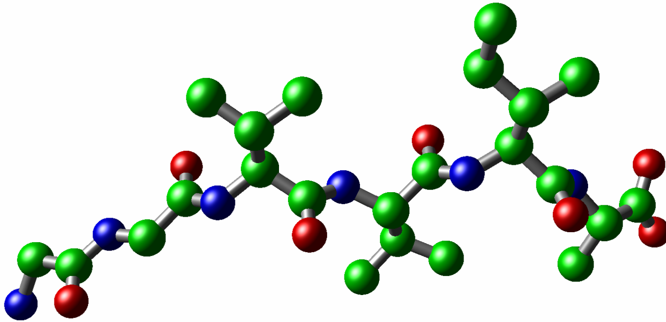

Figure 12: Ball-and-Stick representation of the 2ONV Protein. The used graphical primitives are spheres and cylinders.

Apart from spheres, cylinders are probably the second most-used glyph primitive, especially as connection element, e.g. in ball-and-stick representations for molecules (cf. Figure 12). Therefore the silhouette approximation for cylindrical glyphs is of special interest. Obviously, the generic approach described above of using an object-space bounding box is applicable, but since cylinders are rather simple quadratic shapes, an analytic solution of a tighter fitting bounding geometry is possible.

If the cylinders are rather long and thin using OpenGL *point* primitives might also not be optimal, as they always result in image-space squares. Uploading data for four points spanning a quad, or employing the geometry shader to generate a freely definable quad out of a single point uploaded can be a better solution, depending on the screen-coverage of the glyph and the total number of glyphs to be rendered (Details will be discussed in chapter 2.2). For spheres this approach usually introduces more overhead than the saved fragment processing workload justifies, and for complex glyphs fitting an arbitrarily-shaped image-space polygon is computationally demanding (and even disadvantageous if the number of vertices forming the polygon is not fixed).

Similar to the approach to simplify the ray casting calculations for spheres, by defining a glyph space, in which the sphere is placed at the origin, for cylinders a glyph space can be defined, in which the cylinder is aligned along the x axis, has a radius of 1, and a finite length by cutting off at the distances $x \geq 1$ and $x \leq -1$. The transformation from object space into this *glyph space* can be described by a single matrix. Applying the tangent-plane approach presented for spheres allows for determining lines along the infinite cylinder which are touched by tangent planes, by performing the required calculations in the y-z-plane in glyph space.

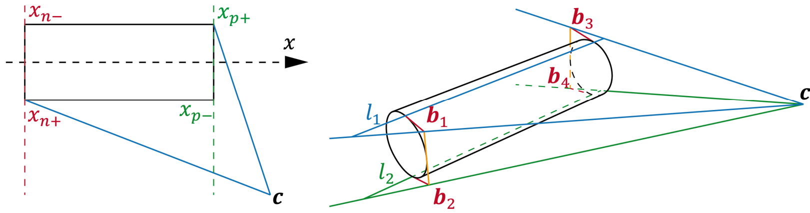

Figure 13: Determining the spanning points for the image-space footprint of a cylinder; Left: near or far points are chosen based on the cameras x coordinate only. Right: The spanning points $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, $\boldsymbol{b}_3$, and $\boldsymbol{b}_4$ are calculated based on the tangent planes (blue and green) on the cylinder going through the camera position and touching the cylinder at the lines l_1 (blue tangent plane) and l_2 (green tangent plane).

Additionally, the cylinder is projected into the plane defined by the x axis and the camera position in glyph space. Since the cylinder is truncated to a finite length, depending on the x coordinate of the camera either the point of the front line or back line of the image of the cylinder will define the extent in image space. These calculations yield the four points in glyph space which will build up a tight image-space quad as footprint approximation.

Figure 13 depicts these calculations. The left figure shows how, based on the camera's x coordinate, the point (implemented only as sign-based factors) from the plane through x axis and camera position are chosen (e.g. using x_{n-} if $c_x < x_n$ with x_n being the negative extent of the cylinder). The right figure shows how, based on these factors, the four points are calculated which finally span the image-space footprint. l_1 and l_2 denote the common line of the cylinder and the two tangent planes through the camera position. The values $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are first calculated in the y-z-plane of glyph space, similar to the calculations for spherical glyphs, and then the final points $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, $\boldsymbol{b}_3$, and $\boldsymbol{b}_4$ are reached following the lines shown in red, which lie within the tangent planes of l_1 and l_2 and move perpendicular to the x axis to the point of maximum extent of the cylinder towards or away from the camera (shown by the orange lines) based on the factors $x_{n\pm}, x_{p\pm}$ evaluated before (left figure). The points $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, $\boldsymbol{b}_3$, and $\boldsymbol{b}_4$ are then projected into image space and form a tight silhouette for the cylinder.

There is one special case that needs to be handled separately which occurs if the camera's distance to the x axis in glyph space is less than the radius of the cylinder. The camera is then either inside the cylinder (in which case it is safe to assume a screen-filling quad) or the camera sees only one cap of the cylinder. In the latter case the four points spanning the image-space silhouette are given by four points spanning the cap (trivial: $\boldsymbol{b}_{1,2,3,4} = (\pm l, \pm r, \pm r)^T$ with l being the length and r being the radius of the cylinder. The sign of l is defined by the x coordinate of the camera position in glyph space).

The quad spanning the image-space silhouette can be used directly if either quad information is sent to the graphics card or if a geometry shader is applied to generate an arbitrarily-placed quad from the uploaded point data. When using the *point* primitive the image-space quad is simply extended to a window-aligned square using min/max-operations. This is an optimization problem whether the additional data upload or the additional overhead of a geometry shader can be compensated by the reduction in fragments. To test this, a generated data set with 500,000 cylinders with a length/radius ratio of 2:1, resulting in a fragment processing overhead of $\approx 50\%$, and a size chosen such that the cylinder glyphs will overlap only slightly was measured on different hardware platforms. The results of older generation hardware were originally published in [GRE09] and [Rei08]. The view port had a size of 512×512 and the data set was zoomed to completely fill out the window, while still being completely visible. These three methods for handling the silhouette approximation were used: a single point (screen-space axis-aligned bounding square), object-aligned quad (data upload increased by $\times$ 4), and point upload and object-aligned quad output from a geometry shader. Note that the geometry shader has to output a much higher number of attributes per vertex than in comparable approaches, e.g. the billboards generation in [LVRH07].

Table 1: Performance (fps) of ray casting cylinders, depending on the chosen bounding geometry approximation method: point-based, quads, or using the geometry shader. Data set contains 500,000 non-overlapping cylinders with length-radius ratio of 2:1.

CPU, GPU	point-based	quad upload	geometry shader
Core2 6600, 88 GTX	61.6	22.5	20.7
Core2 6600, GTX 280	97.1	22.4	248.9[2]
Core2 6600, GTX 480	74.7	18.0	52.8
Core2 8400, GTX 560	130.8	28.0	91.0
i7 x980, GTX 280	82.9	29.4	57.9
i7 x980, GTX 580	211.8	60.0	160.1

The results of the conducted tests can be seen in Table 1. The ray casting calculation is rather cheap. However, using quads to render the glyphs not only increases the data upload by a factor of four, which is rather insignificant for such a small data set, it also increases the work load for the silhouette approximation. As this computation is carried out in the vertex shader stage, it has also to be performed four times for each glyph. Altogether, the quad method can be regarded as a compatibility implementation, as the geometry shader performs acceptably on modern GPUs. Nevertheless, the brute force point-based approach is still significantly faster, as it comes with the smallest overhead. These results clearly recom-

[2] This value was most likely measured incorrectly. However, the required hardware to repeat exactly this test is no longer available.

mend using the point primitive, even if many fragments will be wasted. If this is not possible, e.g. point primitives are known to not working correctly with AMD/ATI hardware, the use of a geometry shader is the best alternative, especially on current GPUs.

However, these results strongly depend on the tested case. For other glyph shapes, which might be more costly to ray cast, the results might favour a different approach. E.g. Klein [Kle08] measured the performance impact of tight-fitting polygonal silhouettes for ellipsoid glyphs. Although the ellipsoids have a similar size ratio as the cylinders used in the test above, Klein found that hexagonal silhouettes computed in the geometry shader were optimal for his data sets.

2.2 Data Transfer

Using the point-based glyph visualization described, the data required on the graphics card is reduced to the necessary minimum. Assuming that, at least for now, the rendering is no longer a limiting factor, the question arises how to optimize the data transfer to the GPU. There are two relevant issues: the data transfer from the secondary storage into main memory and the final upload of the data into the graphics card memory. Simulations of interesting scenarios typically require at least hundreds of thousands of particles and trajectories with at least hundreds of time steps. This amount of data exceeds the available memory of graphics hardware and even poses a significant challenge for the main memory on common workstations. Hence the data needs to be streamed as fast as possible through both transfer points which easily become severe bottlenecks.

The problems at both points can be tackled with two approaches: optimizing the hardware utilization and reducing the data load by applying some sort of compression. While the compression, although restricted by the possibilities of the corresponding processors and their available compute power, is basically the same for both transfer buses, the possibilities of hardware utilization differ completely.

2.2.1 Graphics Hardware Bus

The graphics hardware bus, nowadays PCI-E, offers good bandwidth of ~15.4 GB/s (PCI-E 3.0 x16). However, using the graphics APIs optimally to reach the possible peak performance is quite challenging. The OpenGL API offers a wide range of different functions for data upload and rendering, ranging from immediate mode, infamous to result in worst performance possible, to different forms of buffer objects, allocated and stored in graphics memory. There was an extensive evaluation of the performance of these functions in the context of MD data sets performed and published in [GRE09] assuming the scenario of visualising time-dependent data

sets. The focus lies on the data transfer between CPU and GPU and the streaming of the data from secondary storage is excluded for the moment.

One problem when measuring data upload performance is the high degree of interdependency of the data transfer and the rendering itself. Different ways of data upload will result in different ways the rendering can be performed. It is thus impossible, or at least unfair, to measure one aspect alone, as e.g. latencies in the data upload can be hidden by correctly scheduling rendering commands. Thus, the overall performance is always measured instead. The measurements include setups where e.g. rendering load is negligible, as lower bounds, and ones where the data transfer is excluded altogether, as upper bound for the rendering performance.

The first point to be optimized is the data layout in memory. There is much work on optimized in-core representations and hierarchical data structures, e.g. [RL00] and [SPL04] are two examples among many others. Linear memory layouts were studied and appreciated, not only for their benefits for rendering performance, but also for the advantages for out-of-core rendering [GP07]. However, most of the time the authors simplify the problem by assuming the data fits into GPU memory and thus ignoring the problem of the data upload altogether. Of course, using static VBOs will result in best possible performance, apart from more advanced culling or rendering methods, but this approach is simply not possible for large time-dependent data sets or, even worse, for in-situ visualization of running simulations. The transfer performance of texture data has been analysed in [ED06]. There is also a more generic graphical benchmarking utility available [BFH04], but it does not cover the aspects of vertex data upload and point-based visualization for particle data sets.

However, treating data upload and rendering as interdependent and as a single optimization problem results in two drawbacks. First, the performance tests are similar to *black box* tests, meaning there is no detailed information on which issued command has which impact on the performance or where parts of the computer might get stalled. Second, the upload times, i.e. values in milliseconds, cannot be given making comparison with other publications hard. To provide context to the measurements, as mentioned above, two tests will show the upper and the lower bound of the performance. The immediate mode rendering relies highly on the CPU and the data transfer bus and can thus be assumed to perform worst, as long as the GPU computations do not become the bottleneck. For the upper bound we follow the assumption that the whole data set can be stored in GPU memory using static VBOs, removing the data upload altogether. For an analogous elimination of the GPU computation load, the tests generate a single fixed-coloured fragment per vertex instead of using a sphere ray casting shader.

The tested upload mechanisms are the OpenGL *immediate mode*, *vertex arrays* (VAs), and VBOs with different options. Table 2 summarizes the different methods, the important OpenGL functions, and the employed settings. The data sets to be used in these tests were all generated from a statistical random distribution and consisted of 10^6 and 10^7 particles either ray casted as spheres or splatted as

single fragments. The radii of the spheres were chosen such that the particles would not overlap too much, as the fragment depth replacement has been observed to have an impact on the rendering performance.

All tests were conducted several times and the results were averaged after removing outliers. The performance values are given in fps, which were measured after the whole data set was loaded into main memory. The view port had a size of 512×512 pixels and the data set was zoomed to use the whole view port while still being completely visible. Please note that not all tests were run on all machines. Especially the 10^7 data set was measured much later after the other tests had been conducted and the hardware used for the original tests was no longer available. The appendix of the original publication also includes further performance values, which are omitted here, as these GPUs are no longer relevant.

Table 2: Explanation of the different uploading mechanisms used in the performance tests

<table>
<tr><td>Name (short name for other tables) Comment</td><td>OpenGL Calls
Description</td><td>Main Parameter</td></tr>
<tr><td rowspan="2">Immediate (im.)
lower-bound reference</td><td>glBegin
glVertex+</td><td>GL_POINTS</td></tr>
<tr><td colspan="2">Manual upload of individual data points</td></tr>
<tr><td rowspan="2">Vertex Array (VA)</td><td>glVertexPointer
glDrawArrays</td><td>GL_POINTS</td></tr>
<tr><td colspan="2">Direct array data upload</td></tr>
<tr><td rowspan="2">VBO static (VBO stat)
upper-bound reference</td><td>glBufferData
glDrawArrays</td><td>GL_STATIC_DRAW
GL_POINTS</td></tr>
<tr><td colspan="2">Reference rendering with only one upload (not time-dependent)</td></tr>
<tr><td rowspan="2">VBO stream (VBO strm)</td><td>glBufferData
glDrawArrays</td><td>GL_STREAM_DRAW
GL_POINTS</td></tr>
<tr><td colspan="2">Buffer object upload meant for data “modified once and used at most a few times”[3]</td></tr>
<tr><td rowspan="2">VBO dynamic (VBO dyn)</td><td>glBufferData
glDrawArrays</td><td>GL_DYNAMIC_DRAW
GL_POINTS</td></tr>
<tr><td colspan="2">Buffer object upload meant for data “modified repeatedly and used many times”[3]</td></tr>
<tr><td rowspan="2">VBO dynmapping (VBO dynmap)</td><td>glMapBuffer
glDrawArrays</td><td>GL_WRITE_ONLY
GL_POINTS</td></tr>
<tr><td colspan="2">Buffer object memory mapping when CPU memory layout is not optimal</td></tr>
</table>

[3] Cited from the OpenGL SDK Documentation of glBufferData: (last visited: 09.02.2012) http://www.opengl.org/sdk/docs/man/xhtml/glBufferData.xml

Table 3: upload performance (in fps) for rendering 10^6 particles as points (upper part) or ray castes as spheres (lower part)

CPU, GPU	im.	VA	VBO stat	VBO strm	VBO dyn	VBO dyn-map
point splatting						
Core2 6600, 88 GTX	15.3	108.0	173.6	29.3	29.2	45.4
Core2 6600, GTX 280	13.7	110.8	292.6	37.9	37.7	44.5
Core2 6600, GTX 480	14.7	61.1	651.3	32.8	32.9	53.8
Core2 8400, GTX 560	3.4	138.1	618.0	41.3	41.7	59.7
i7 x980, GTX 280	4.8	179.6	280.7	65.0	65.1	74.1
i7 x980, GTX 580	4.9	277.5	856.2	63.5	63.9	66.9
Xeon, Q5000	175.0	812.4	865.1	46.4	46.0	108.3
sphere ray casting						
Core2 6600, 88 GTX	15.2	44.7	44.8	29.2	29.1	34.9
Core2 6600, GTX 280	13.8	82.8	82.9	37.1	38.1	n/a
Core2 6600, GTX 480	15.6	60.5	155.5	32.7	32.9	n/a
Core2 8400, GTX 560	3.3	146.3	168.7	42.5	42.2	61.5
i7 x980, GTX 280	4.8	72.0	84.8	65.2	63.3	50.2
i7 x980, GTX 580	4.9	220.1	224.2	63.5	64.1	58.2
Xeon, Q5000	92.0	92.8	91.6	45.4	45.3	93.0

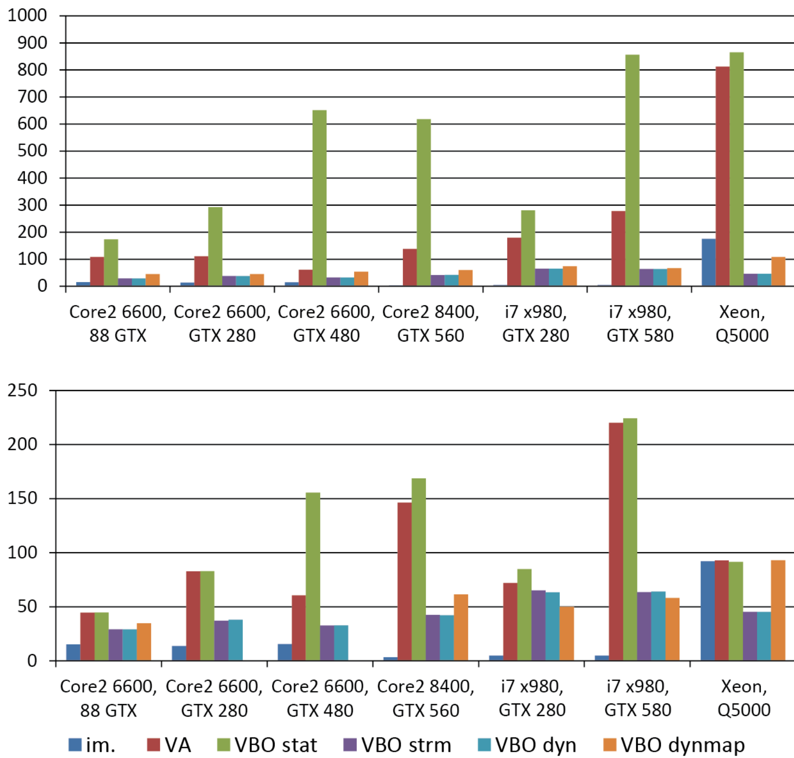

Figure 14: upload performance (in fps) for rendering 10^6 particles as points (upper diagram) or ray cast as spheres (lower diagram)

Table 3, Table 4, Figure 14, and Figure 15 show the results of the upload performance for different combinations of CPUs and GPUs, two data set sizes and the different upload mechanisms. All values are averaged over multiple runs. Please note that small differences in the frame rates may lie within the error boundaries acceptable for the measurements. For example, Table 3 shows for rendering 1M spheres on the *i7 x980, GTX 280* hardware 65.2 FPS for VBOs in *streaming* mode and 63.3 FPS in *dynamic* mode. The actual measured intervals are 57,92 – 65,25 FPS for streaming and 60,64 – 65,30 FPS for dynamic VBOs.

Table 4: upload performance (in fps) for rendering 10^7 particles as points (upper part) or ray cast as spheres (lower part)

CPU/GPU	im.	VA	VBO stat	VBO strm	VBO dyn	VBO dyn-map
point splatting						
Core2 8400, GTX 560	0.09	14.42	76.98	0.43	0.43	7.45
i7 x980, GTX 280	0.13	15.79	29.61	6.36	5.99	6.33
i7 x980, GTX 580	0.13	31.58	109.81	6.17	6.39	8.30
sphere ray casting						
Core2 8400, GTX 560	0.07	16.13	24.53	3.59	3.56	6.89
i7 x980, GTX 280	0.13	9.06	11.79	6.12	6.37	5.89
i7 x980, GTX 580	0.13	31.66	34.18	6.25	6.40	8.30

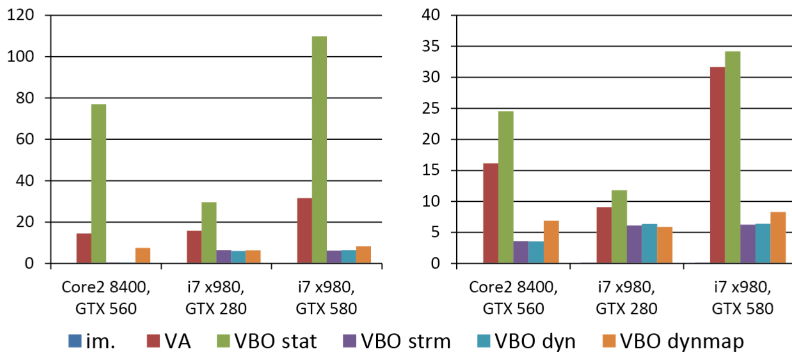

Figure 15: upload performance (in fps) for rendering 10^7 particles as points (left diagram) or ray cast as spheres (right diagram)

Rendering with static VBOs is included as upper bound, as is does not include any data upload time but only solely the rendering performance. On the contrary, the immediate mode upload is highly CPU focused and requires a tremendous amount of drawing calls. Only on a very high end machine (Xeon CPU with Nvidia Quadro 5000 GPU) this workload can be handled acceptably. However, it is note-

worthy that Quadro cards come with different drivers than GeForce cards, which is likely to impact the performance results.

Throughout all tests, the particle upload using VAs results in the best performance. Only – again on the high end system – if the graphics card's computation is the bottleneck, the VBO's performance (dynamic mapping) is capable of reaching the performance of VAs. In all other cases VAs clearly outperform any of the VBO methods. In this context it is especially unfortunate that VAs were marked as deprecated in OpenGL 3 [Opeb] and are removed from the core profile of OpenGL 4. As for the different options available for the VBOs, there does not seem to be a difference, at least not for the tested application. This, however, is not true for the possible access methods. Uploading data into a VBO is clearly inferior to mapping a VBO into main memory, as is shown by the example of dynamic VBOs. One explanation for the poor performance of the VBOs compared to the VAs might be the asynchronous nature of the evaluation of graphics commands. As such, it is likely that when using VAs the memory containing the data is pinned, the data transfer is started, and as soon as the very first vertex information is available on the GPU the rendering is started in parallel. In contrast, the VBOs represent buffers in GPU memory. The data transfer and the drawing calls are separated in this case, meaning that most likely the whole data has to be transferred into GPU memory before rendering can actually start. This argument also provides a clue on which mechanisms to be used in future, when VAs are no longer available: the data upload and rendering must be manually interleaved, optionally using multi-threaded OpenGL. Either only part of the data are uploaded and rendered concurrently, or the data required to render one image is stored and used on the GPU while the data for the next images is already being uploaded, utilizing frame-to-frame coherency, assuming there is enough graphics memory.

These ideas of interleaving rendering and uploading of the data or parts of the data also offer the possibility to optimize rendering when the data is not or only partially changed between consecutive rendered frames, as this data can be cached in VBOs in graphics memory. Even if the whole data set does not fit into graphics memory, at least parts of the data required multiple times might fit in, and even this will increase the rendering performance as the corresponding data upload latencies are removed. However, since the performance measurements above showed that the data transfer into VBOs is significantly slower than the data transfer with VAs it can be expected that exclusively using VBOs will result in strong performance variations and long latencies as data needs to be updated. To avoid this problem the data upload into this VBO-based cache should be limited per frame, which is especially recommendable in a single-threaded OpenGL application. If more data needs to be transferred to the GPU VAs should be used and the upload of that data into the VBO-based cache is postponed until one of the next rendered frames. This strategy is obviously not applicable when streaming data from time-dependent data sets or performing in-situ visualization of fast simulations, as the data to be transferred to the GPU is always changing and thus never reusable.

Table 5: upload performance values (fps) for uploading quantised positional data.

	CPU, GPU	Core2 6600, 88 GTX	Core2 6600, GTX 280	Core2 6600, GTX 480	Core2 8400, GTX 560	i7 x980, GTX 280	i7 x980, GTX 580
1M Floats	VA	118.5	121.7	63.9	148.7	192.0	306.9
	VBO stat	250.0	292.6	670.0	471.8	281.0	720.5
	VBO dyn	69.0	68.4	114.8	123.4	140.0	208.2
4M Floats	VA	31.0	31.5	16.5	39.1	49.2	82.3
	VBO stat	69.2	74.0	190.8	135.4	73.9	211.3
	VBO dyn	17.9	17.5	27.5	33.6	36.1	54.5
1M Shorts	VA	222.6	234.6	121.1	275.7	234.9	561.1
	VBO stat	249.9	292.5	676.6	473.0	280.5	728.5
	VBO dyn	125.7	141.8	194.4	220.4	187.1	323.2
4M Shorts	VA	60.0	61.0	31.9	82.1	61.2	155.8
	VBO stat	69.2	74.1	193.2	135.7	73.9	213.6
	VBO dyn	35.6	36.2	50.3	62.9	48.4	86.7
1M Bytes	VA	219.1	228.8	141.3	291.8	201.2	353.2
	VBO stat	250.0	292.7	675.6	473.4	280.9	726.8
	VBO dyn	167.6	198.2	303.8	324.7	224.2	447.1
4M Bytes	VA	58.8	59.9	31.6	79.8	55.0	93.6
	VBO stat	69.2	74.1	192.8	136.0	73.9	213.6
	VBO dyn	47.5	50.0	79.7	90.0	58.4	123.5

In that case different methods for data compression can be used to lessen any data transfer bottleneck. One of the simplest but most effective mechanisms is data quantization. Several quantization strategies for geometry and attributes have been presented to date, e.g. hierarchical positional quantization [HE03], uniformly distributed normal vectors [Paj03], or colour, geometry, and normal quantization for point rendering [RL00]. To test one quantization scenario, measurements were conducted, quantizing positional data from floats to shorts or bytes. Two data sets were tested, one with 1 million particles and one with 4 million particles (because 1 million floats are the same amount of data as 4 million bytes). It is assumed that a positional hierarchy, similar to [HE03] is used to reduce the positional error introduced by the loss of resolution through quantization. Again, the data set is splatted into a 512×512 viewport as constant coloured single fragments to exclude performance effects from different rendering techniques. The data upload is performed by VAs or dynamic VBOs. Performance values of rendering with static VBOs (no data upload per frame) are included as reference values.

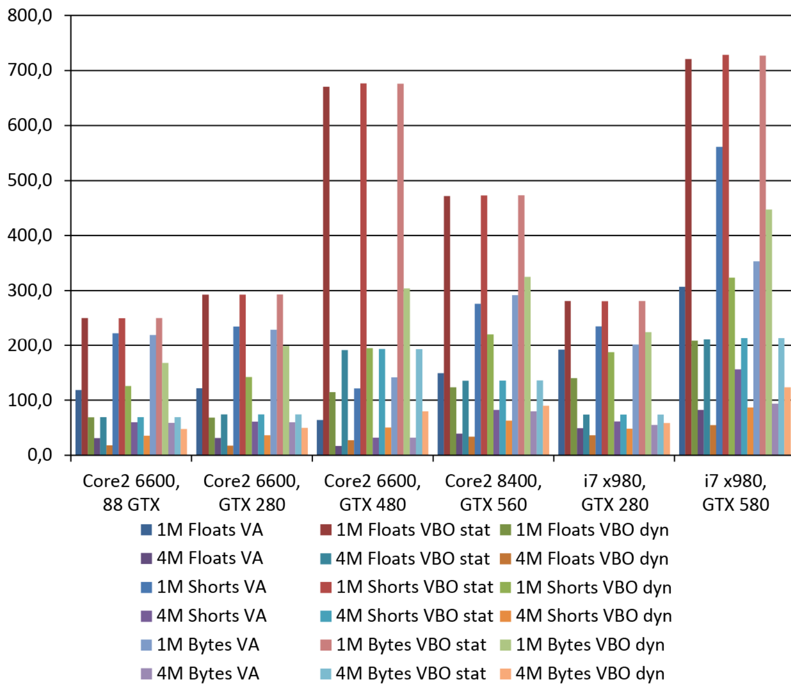

Figure 16: upload performance of quantised positional data in fps. VBO static values (no upload) included for rendering performance reference of the corresponding hardware

The results of the performance test measurements for quantised positional data are shown in Table 5 and Figure 16. The quantization almost always has the expected positive effect on performance, especially when data transfer via dynamic VBOs is used. However, not always are the speedup factors reached which should be theoretically possible; i.e. a speed-up of × 4 when using bytes instead of floats. It is especially noteworthy that the quantization to bytes often does not produce a significant performance benefit compared to using shorts. In some cases the performance instead drops drastically. Similar to the results from the original publication [GRE09] this yields the conclusion that quantization to bytes is almost not relevant from a performance point-of-view. Apart from this aspect quantization is of course reasonable to save GPU memory. Of course, as mentioned above, to remedy the problems introduced by the loss of resolution through quantization additional measures, like a positional hierarchy of relative coordinates, are required, which will introduce additional overhead, but which might not have a too critical impact.

Thus quantization cannot be recommended as a general tool to increase performance. Instead it highly depends on the task at hand whether or not quantization can be applied. For example, using a positional hierarchy of relative coordinates [HE03] to work with quantized positions makes interpolation between two adjacent time frames far more difficult. Either the hierarchy must be constant over the complete trajectory (similar to the approach in [HLE04]) or the interpolation of the positions must take place between the relative coordinates of different hierarchies. Data sets with PBCs aggravate this problem. Other approaches to compress the data also exist, but always introduce significant additional overhead. Especially the decompression of the data on the GPU limits the possibilities, as the features of GPUs, although steadily increasing, still do not offer the same flexibility as a CPU. Nevertheless, the presented strategies for handling the upload of particle data from main memory to the GPU allow for fast transfer and interactive rendering of large data sets.

2.2.2 Loading from Secondary Storage

The second data transfer bottleneck is the loading of data from secondary storage. Especially for time-dependent data sets we cannot assume that the whole data set can be stored in the main memory of the machine. Even high-end machines will reach their storage limits eventually. Practice shows that data transfer from secondary storage should be organized in rather large, continuous blocks of data. In addition the question arises how to optimize based on the different file loading mechanisms and caching strategies. In this context the caching mechanisms of the operating system or the hardware itself are of no special interest, because they cannot be influenced by an implementation anyway. Therefore, the focus lies on how a data set is loaded, especially when seeking within a time-dependent data set, which requires random access.

There are different options to choose from for an actual implementation: most importantly the different APIs for data transfer. The experiment described in the following was conducted to identify benefits and problems for the given scenario. Two test files were generated which resemble the structure of an optimized file format for real data. The data sets are built up from data blocks (4 million and 4 thousand) of fixed size (4 kB and 4 MB); such that both data sets have roughly the same size of 16 GB. As a test for sequential reading the data sets are completely read. A second test to analyses random access uses a generated index file storing the order in which the blocks of the data file need to be read. This way, it is ensured that the whole data set needs to be loaded, excluding any caching effects from blocks being read several times, but due to the non-sequential order the access pattern resembles real random access. The tests were conducted on Fedora Linux 14 (Laughlin) in an interactive X11 session, however, with only a single user logged in. The test machine is an Intel I7 CPU (3.33 GHz), an Asus P6X58D-E main board, with 12 GB

DDR3 RAM (1333 MHz), and the files were loaded from a conventional hard disk Western Digital RE3 (500 GB) and a solid-state disk Intel X25-M G2 (80 GB). The tested APIs were the Posix functions (e.g. *fopen*[4]), IFStream[5] implementations of C++ (often used for their easy-to-use and flexible interface), an implementation based on the low-level file functions (e.g. *open*[6]), and an implementation for *memory mapped file IO*[7] which should in theory perform best due to the utilization of DMA data transfer. Table 6 shows the resulting transfer speeds of the experiment (in MB/s).

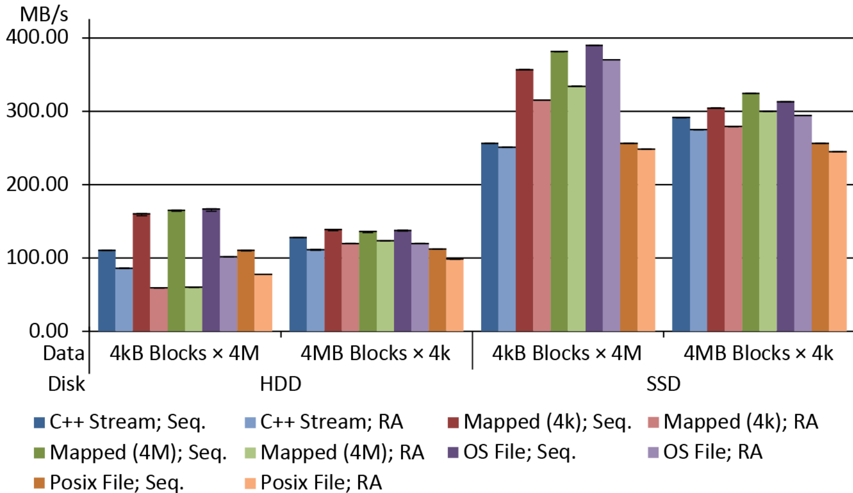

Figure 17: Read performance from secondary storage; all values are median values over 12 measurements and are given in MB/s

[4] http://www.cplusplus.com/reference/iostream/ifstream/ (last visited: 19.01.2012)
[5] http://linux.die.net/man/3/fopen (last visited: 19.01.2012)
[6] http://linux.die.net/man/2/open (last visited: 19.01.2012)
[7] http://linux.die.net/man/2/mmap (last visited: 19.01.2012)

Table 6: Read performance from secondary storage; all values are median values over 12 measurements and are given in MB/s.

API	Disk / Data Mode	HDD		SSD	
		4KB Blocks × 4M	4MB Blocks × 4K	4KB Blocks × 4M	4MB Blocks × 4K
C++ Stream	Seq.	110.36	127.74	256.14	291.34
	RA	86.09	111.36	250.83	274.79
Mapped (4K)	Seq.	160.27	138.59	356.70	304.27
	RA	59.43	119.67	315.07	279.17
Mapped (4M)	Seq.	164.85	135.96	381.32	324.37
	RA	60.07	123.37	334.07	299.65
OS File	Seq.	166.68	137.57	389.72	313.07
	RA	101.56	119.71	370.10	294.32
Posix File	Seq.	110.21	112.21	256.31	256.36
	RA	77.56	98.73	248.50	244.96

Each measurement test was performed 12 times. The highest and lowest values were regarded as outliners. Figure 17 shows the median values of the remaining results. Some findings are conform to expectations, e.g. that the impact of the seeking operation during the random access tests is higher for the data set constructed of 4 kB sized blocks, as simply the number of blocks and thus the number of required seek operations is higher. Because of the large file size, minimizing the effects of caching, and the isolated testing environment, minimizing the effects of other processes accessing the secondary storage, the results vary only minimally, as is shown by the almost non-visible error bars in Figure 17. However, there are also several surprising results: reading 4 MB data blocks results in worse overall performance than reading the same amount of data as 4 kB data blocks. Furthermore the performance of the Posix file (red) is rather low, considering it should only be a thin abstraction around the operation system file functions (orange). Then again, the performance of the C++ streams, which are likely to come with huge overhead, perform quiet well, especially when reading large blocks, even when doing seek operations. But most surprising of all was the fact that the memory-mapped file IO did not produce any benefit compared to the classical file IO. A possible explanation would be that the implemented reading pattern hinders the DMA to be beneficial, as the data is requested in a blocking fashion, which, however, is the common case for loading data sets, while DMA could benefit from asynchronous operations: i.e. requesting a mapping of a portion of a file, and then doing something else before accessing the data to conceal the transfer latencies. To give a suggestion as conclu-

sion from this test, using the operating system file functions is recommendable, as this is the API with the best average performance.

Now implementing a simple streaming of data into main memory is rather simple. First, the memory footprint of a single time frame needs to be calculated. For data sets with changing particle counts per time frame or otherwise applied means which will result in different memory requirements per time frame a compromise between a conservative maximum estimation and the average memory requirement should be used. A simple but acceptable variant would be loading the first time frame, the last, and one from the middle of the trajectory and averaging over these three. Based on the available main memory a buffer is allocated capable of holding the data of several time frames. The loading of the data itself must now be performed in a multi-threaded environment, requiring these data buffers as well as their index structure to be locked against race conditions, e.g. using mutexes or critical sections. The visualization and rendering thread can now request a data time frame from this software cache mechanism. If the requested frame is not readily available from the buffers, either the closest match, in terms of temporal distance, is returned or the visualization thread is blocked until the data is available. The returned data block is locked as long as it is used by the rendering to avoid overwriting by the loading thread. This thread loads the requested time frames and pre-fetches time frames likely to be requested next, i.e. considering animation playback of the trajectory the time frames following the currently requested data are likely to be requested next. Similarly the temporal distance of the data stored in buffers and the last requested time frames give a good indication on which buffer to overwrite as soon as all of them are filled with data. Of course, this basic schema has to be extended when allowing multiple consuming threads to use the cache, but the basic principles remain the same.

The last issue for optimizing the data transfer is the file format of the data files on secondary storage. The file formats used by simulations or analysis tools are often ill-fitted for interactive visualization. The main concept here is to already store the data in a way which is optimal for the graphics card upload to minimize the required changes of the data in memory as well as the parsing effort, which is especially high if the data file format is based on a textual representation. Most data file formats focus on the analysis process and e.g. provide additional data important for that task, which however, will be dead payload for the visualization and should be removed. Thus a file format conversion as pre-processing for the visualization might be a better solution. A visualization-centric file format should also contain information helping to seek within the file supporting the loading thread. The particle data itself should be pre-sorted to minimize the graphics API stage changes, e.g. shader switches, and to allow block transfer to the GPU. Additional data for analysis tasks should be stored behind the block of data required for rendering using index values for cross-referencing. The actual data layout, of course, depends on the concrete application.

2.3 Occlusion Culling

Given the potential of glyph ray casting and data transfer is depleted with the methods already described, to interactively visualize ever bigger data sets the only option left to shift the limits on normal workstations even further lies in optimizing the rendering process as a whole. Comparing the data set sizes with the limited resolution and size of computer displays, obviously the particle/pixel ratio suggests that there will be many unnecessary calculations resulting in fragments which will be overwritten or which would lie outside the viewing frustum. Considering a standard computer display has roughly 2 million pixels (e.g. Full HD: 1920 × 1080 pixels), only that many particles can be visible at most. Transferring and ray casting that many particles does not pose a challenge for current hardware (cf. chapters 2.1 and 2.2). So the optimization strategy is to remove as many particles as possible from the active data which will not result in visible pixels anyway. This is the idea of the computer graphics concept of culling.

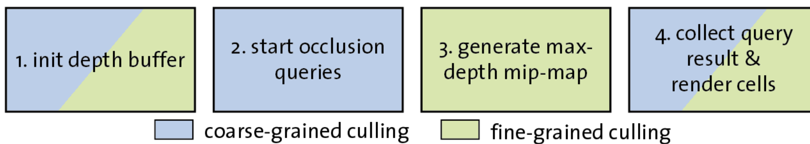

Figure 18: Schematic process steps of a two-stage culling method for particle-based data. Generating the depth mip-map in stage 3 is required for fine-grain culling during stage 4 and hides the latencies introduced by the occlusion queries issued in stage 2.

This method was originally presented in [GRDE10]. The main concept is to use a two-stage culling to reduce both the data transfer as well as the rendering load. Figure 18 summarises the different stages of this culling technique. The data transfer load is reduced by determining visibility of spatial subvolumes of the data using hardware-supported occlusion queries (HWOQs), while the rendering is optimized using a replacement for the unavailable early depth test using a hierarchical maximum-depth mip-map (HZB). Current GPUs in principle already support depth tests with HZB natively [GKM93]. However, glyph-based ray casting disables this mechanism by having the fragment shader overwrite the depth value estimated by the rasterization stage. Both of the proposed culling approaches utilize a frame-to-frame coherency strategy to minimize the overhead for the additional tests required. This approach is closely related to the deferred splatting method [GBP04], in which the visible point primitives from one frame are used to initialize the depth buffer for the next frame using cheap point primitives as estimate to finally save costly operations for high quality surface splatting of invisible primitives.

The latencies introduced by the HWOQs are hidden by performing operations for the HZB in parallel, optimizing the GPU utilization. Other strategies to

reduce the overhead introduced by the latencies of HWOQs have been developed. Guthe et al. [GBK06] modelled occlusion probability using statistical means, and by this aim reduced the number of unnecessary queries. Bittner et al. [BWPP04] and Mattausch et al. [MBW08] use occlusion queries to perform culling in complex scenes where a bounding volume hierarchy is available.

There has been a huge amount of research in the field of computer graphics on determining the visible and occluded parts of scenes. Comprehensive overviews were given by Bittner and Wonka [BW03] and Cohen-Or et al. [COCSD03]. As the presented culling algorithm focuses on the particles which will be visible in the end HWOQs, which are supported by current GPUs and which allow to determine the number of visible fragments created by rasterizing the geometry against the depth buffer, provide a perfect base mechanism. Further methods to determine the visibility exist, like the HZB already mentioned or variants of this method as in [Déc05]. The item buffer approach, presented by Klosowski and Silva [KS00], which was later extended by Engelhardt and Dachsbacher [ED09], uses unique colours for each object to determine their visibility by counting pixels of the corresponding colour after rasterization. Similar to the HWOQs recent GPUs provide predicated or conditional rendering [Opea] in which a query is issued by the application, and afterwards the GPU is instructed to render geometry only if at least one fragment of the original query remained visible after depth test. This method, however, does not reduce the data transfer load.

The presented two-level occlusion culling builds on the following stages which are also outlined in Figure 19:

1.: Initialize the depth buffer for occlusion culling rendering cheap conservative depth estimates for the particles which were visible in the last frame.

2.: Issue HWOQs for all cells' bounding boxes of the spatial data structure.

3.: Compute a HZB from the depth buffer of step 1 for fine-granular culling.

4.1: Read back the results of the HWOQs, update the list of visible particles, and render all visible glyphs with per-glyph culling directly on the GPU.

4.2: Perform deferred shading. Details on this stage are given in Chapter 2.4.1.

The HWOQs required a coarse subdivison of the data. The presented approach uses a regular grid as spatial structure to organize the particle data. A hierarchical data structure is not reasonable, because it would make a stop-and-wait algorithm necessary [MBW08], which in turn would require HWOQs to be interleaved with the rendering. Additionally, despite the simplicity and hence possibly higher number of HWOQs with a regular grid, the performance results show no detrimental impact.

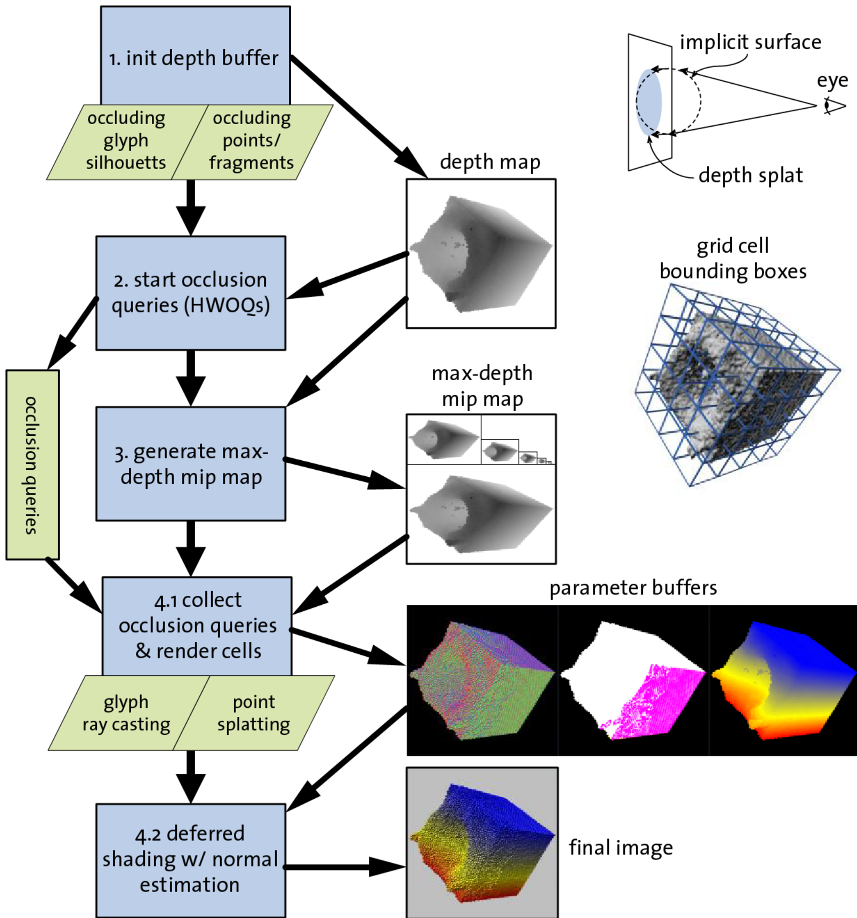

Figure 19: Details on the stages of the culling method: 1. Initialization of the depth buffer with known occluders from the previously rendered frame; 2. Start of HWOQs for all grid cells by testing their bounding boxes; 3. Generation of HZB; 4.1. Collection of results of HWOQs, update of the list of visible cells, and rendering of visible glyphs. Stages 1 and 4.1 can output ray cast glyphs or single flat-shaded fragments if the glyphs become too small in image-space. Stage 4.2 implements deferred shading and is described in detail in chapter 2.4.1. Note that the rendering in stage 1 initializes the depth buffer with a conservative depth splat for the HZB, as well as for subsequent render passes.

Step 1 of the presented method renders all particles from the grid cells marked as previously visible into the depth buffer for the current frame. The resulting depth buffer will be used to perform both types of culling. For these glyphs, the ray casting algorithm is not performed in its complete form. Instead only a conservative maximum depth estimate is calculated in the vertex shader and is not changed in the fragment shaders, thus enabling the hardware-supported early depth test. The efficiency of the depth test is further enhanced by sorting the cells to be rendered from front to back, which can be carried out very fast by using a stable sorting algorithm and keeping the order across frames. The fragment shader performs only the calculation to generate the precise silhouette of the glyph.

In stage 2 the HWOQs are issued against the created depth buffer by rendering the bounding box geometries for all cells of the grid. Cells previously invisible might now become visible and vice versa because of changes in particle densities within the occluding cells or changes of the view parameters. The performance of this stage depends on the amount of issued HWOQs which result from the resolution of the spatial grid. On the one hand, more cells will yield more accurate results for the cell-level culling, but on the other hand, will also create more queries and thus longer latencies until the query results are available for stage 4.1. For the data sets presented here (cf. Table 7) and in the original publication a grid of 15^3 cells was sufficient to get satisfactory results, as evidently stage 3 is successfully capable of hiding the latencies of the corresponding 3375 HWOQs. This is a trade-off as a finer grid would possibly allow removing more cells from further processing in dense regions of the data, but would also introduce more queries and thus more latencies.

Table 7: Sizes and descriptions of the example data sets: D1 – D4 have been created by molecular dynamics simulations, D5 is an artificial data set and has been created using a statistical distribution.

data set	number of glyphs	description
D1	107,391	small laser ablation
D2	4,456,963	small crack propagation
D3	44,569,630	large crack propagation
D4	48,000,000	large laser ablation
D5	100,000,000	artificial test data set

The task of coarse-grain culling on cell-level is completed in stage 4 in which the results of the HWOQs are read back from the graphics hardware. The visibility flags of the cells of the spatial grid are then updated correspondingly and the particles of the visible cells are finally rendered using ray casting. Using a conservative estimate based on the particle size and cell placement relative to the view point, the particles are either rendered using a complete glyph shader or a shader performing a cheap splatting of single fragments. The latter one is used for particles whose glyphs would be of sub-fragment size anyway. The remaining calculations are

performed in step 4.2 which will be described in chapter 2.4.1. This concludes the coarse-grain culling reducing the data transfer.

For the fine-grain culling, which is meant as replacement for the disabled early depth test, the HZB needs to be computed. Stage 3 performs this calculation based on the depth buffer created in stage 1. A mip-map pyramid is created via Ping-Pong rendering with two frame buffer objects (FBOs), down-sampling using the maximum of 2×2 values for the next mip-map level.

As stated above, when performing the rendering of the particles in stage 4.1 the image-space size of the particle is estimated in order to decide whether to perform full ray casting or to only splat single fragments. This estimation also yields the mip-map level we need to query whether the glyph is occluded or not. In the correct level, the glyph's image-space footprint covers four texels at most. The maximum value of these is compared to the minimum depth of the glyph to decide the glyph's visibility in a conservative way. Glyphs which are definitely hidden are removed in their vertex shader by being moved to infinity. While the geometry processing and vertex shaders are still performed for all glyphs transferred to the graphics card, the costly ray casting in the fragment shader and the rasterization is only performed for the particles which are potentially visible.

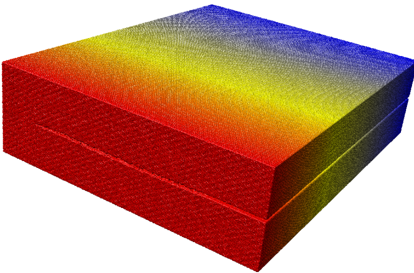

Figure 20: MD simulation D3 of crack propagation in solid material; 44.6 million atoms

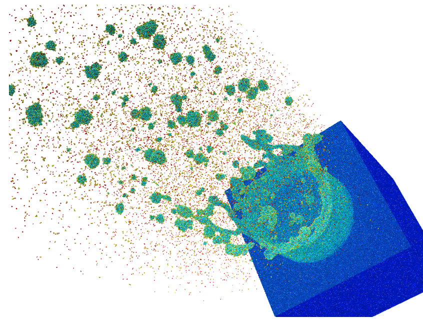

Figure 21: MD simulation D4 of laser ablation; 48 million atoms

This approach is primarily suited for dense data sets. With sparse data sets the potential for performance improvements by culling methods, apart from trivial frustum culling, is very small, although the initial argument of overdraw and pixel-to-particle ratio remains valid. However, for most data sets considered in this thesis the approach yields good results. The used test data sets are shown in Table 7. The smallest of the data sets is D1, which is shown in Figure 23. It shows a time frame from a laser ablation MD simulation. The data sets D2 and D3 (cf. Figure 20) show the results of a crack propagation MD simulation. Both data sets show the same data, whereas D2 is only a smaller slice of D3 comprising exactly 10 per cent (in width). The data set D4 shows one time frame from a larger laser ablation MD simu-

lation (cf. Figure 21). Finally, D5 is a synthetic data set created from a statistical distribution of 100 million atoms. To exclude loading times from secondary storage as well as issues from data streaming, all data sets were stored in the main memory of the test machine, which was running Windows 7 with an Intel Xeon 5530 2.4 GHz with a GeForce GTX 285 with 1 GB graphics memory. The rendering is performed using OpenGL and a viewport size of 1024×1024 pixels. The results are shown in Table 8 and Figure 22.

Table 8: Performance measurements of the two-stage occlusion culling method; All numbers indicate fps if not otherwise noted. Different views were used such that glyphs are large enough in image space and rendered as ray cast spheres (S-*), or splatted as simple points (P-*). The viewing directions and distances were changed to obtain a best (*-Best) and worst (*-Worst) case for the culling algorithms (i.e. maximum/minimum number of grid cells occluded). The last two columns show statistics for the case that both culling levels are active: the percentage of the grid cells that are visible and the number of glyphs that are actually ray cast after the vertex culling stage. Column 3 (culling: none) shows the baseline performance for the un-optimized approach. Columns 6 and 7 demonstrate the impact of caching.

data set	view config.	culling					visible data	
		None	Cell	Vertex	Both	Both	Cells %	# Glyphs
caching		No	Yes	Yes	Yes	No		
D1	S-Best	28.21	76.06	172.67	90.23	89.61	8.09	3600
	S-Worst	32.12	57.25	137.37	68.98	69.88	23.38	3136
	P-Best	621.20	99.42	195.34	97.22	104.70	39.38	27713
	P-Worst	593.23	99.27	200.96	98.33	101.75	45.75	46338
D2	S-Best	4.60	17.96	7.22	42.60	39.62	6.67	12594
	S-Worst	6.92	18.77	12.64	36.30	33.42	20.53	115003
	P-Best	14.17	69.90	24.44	69.81	42.84	6.67	44804
	P-Worst	15.46	46.78	12.91	73.15	65.27	22.93	261221
D3	S-Best	1.29	7.62	1.22	18.27	15.23	6.67	153786
	S-Worst	1.18	3.21	1.06	8.48	6.19	18.70	623188
	P-Best	1.70	16.91	1.62	16.61	9.71	6.70	460425
	P-Worst	1.56	38.02	1.65	38.52	16.53	18.70	989268
D4	S-Best	0.77	1.94	1.23	5.96	4.28	6.67	160066
	S-Worst	0.95	2.33	1.03	7.02	4.44	59.35	654718
	P-Best	1.26	12.41	1.93	12.36	6.56	6.67	214358
	P-Worst	1.19	15.98	1.36	15.43	9.32	58.90	1144420
D5	S-Best	0.88	8.04	0.73	13.65	9.38	6.67	740977
	S-Worst	0.92	2.85	0.55	5.22	3.32	18.70	1313734
	P-Best	1.08	20.71	0.90	20.40	11.70	6.67	799347
	P-Worst	1.10	8.34	0.88	8.26	4.25	18.70	1521383

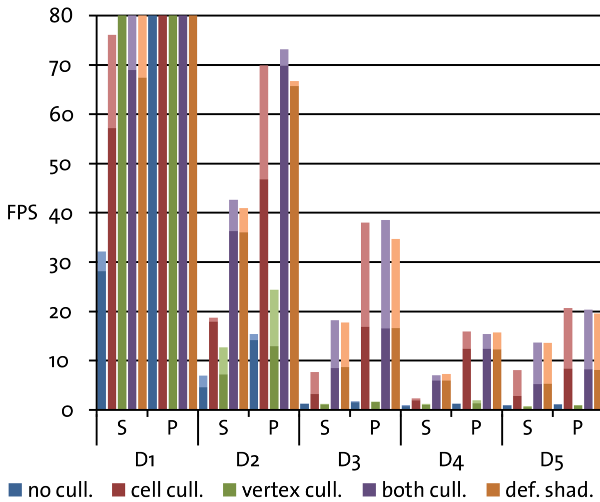

Figure 22: The rendering performance results from Table 8 without any culling (no cull.), cell-level culling only (cell cull.), vertex-level culling only (vertex cull.), both culling techniques together (both cull.), and both culling techniques and a deferred shading pass (def. shad.). Using both culling techniques together results in the best performance. For simple point representations the effect of the vertex-level culling is negligible. This is due to a shift of the limiting bottleneck from rendering to data transfer. The overhead of the deferred shading pass is very small. Note that the bars of data set D1 are truncated to keep the focus on the values of the larger and more interesting data sets.

All data sets were measured only rendering spheres (rows denoted with S-*) and only splatting points (P-*). The performance in real-world scenarios will lie in between these values, because of the automatic switching between these two glyph types.

For all data sets we measured an increased performance using the presented occlusion culling, except for the very small data set D1 when only splatting points. This is due to the introduced overhead compared to the extremely cheap rendering and poor occlusion behaviour, which can be seen by the percentage of the visible cells being 2 to 5 times higher than when rendering spheres. Furthermore, D1 and D2 show unexpectedly low frame rates when rendering spheres (S-*) which is supposedly due to the large overlap of the primitives resulting in many depth buffer replacements. In turn, this provides significant occlusion, causing large performance gains when culling is used.

When comparing the performance of the coarse-grain cell-level culling to the fine-grain vertex-level culling alone, the cell-level culling provides higher per-

formance improvements, which are mainly because of the reduction of data upload. The vertex-level culling can only reduce the workload of the fragment shader. For small data sets, such as D1, the data transfer is not the bottleneck and the vertex-level culling becomes highly beneficial. For large data sets the possible granularity of the cell-level culling is limited and the ray casting cost of sphere glyphs is significant. Here, both culling methods together yield the best performance (e.g. S-* series for data sets D3-D5), which is up to one order of magnitude faster compared to using no culling.

The VBO-based caching strategy further increases the performance by about 40%-50%. As the GPU-side cache is updated incrementally, the strategy is most effective when the viewing parameters change only smoothly. The impact of the caching is most prominent when the workload of the fragment processing stage is not the bottleneck of the entire pipeline.

Typically, deferred shading is used to increase the rendering performance by moving the rather costly lighting calculations from the ray casting stage, which is subject to overdraw, into a single image-space pass. Since our culling strategies almost completely resolve the overdraw problem, this optimization is actually no longer achievable, and in principle, the introduction of the additional image-space rendering pass can even result in a slight performance drop. However, our intention in using deferred shading was not to increase the rendering performance, but image quality. This is achieved by a normal vector estimation which is described in chapter 2.4.1. Additionally, this deferred shading pass allows for shading of the larger spatial structure implicitly formed by single-fragment splats for very small glyphs (P-*).

These results demonstrate the effectiveness of the presented two-stage culling for large data sets. Comparing this implementation to existing tools is sadly almost impossible. TexMol [BDST04] failed to display D2 after successfully loading it but achieved a very good frame rate of about 110 fps for D1. BallView [MHLK05] was able to load D1 and render it at less than 1 fps. AtomEye [Li03] performs quite well for D1, but even for D2 the performance drops to about 0.5 fps. VMD [HDS96] is capable of rendering large data sets interactively using ray casting similar to the methods described in chapter 2.1, but without depth value correction. However, with the default setup, VMD renders D1 at 4 fps and requires more than 20 seconds for a single frame for D2.

2.4 Deferred Shading & Ambient Occlusion

With the optimizations presented in the previous chapters visualizing several millions of particles interactively is possible on a default workstation. However, doing so might not aid the user in the anticipated way, as the resulting images, due to the limited image-space resolution, will suffer from aliasing and visual clutter, and the shape of the data will be hard to perceive. For large data sets the individual particles

will be very small in image-space, resulting in high frequencies. These often do not stem the actual particle positions, i.e. the result in the depth buffer, but the outcome of the lighting calculations due to the high-frequent variation in the surface normal vectors because of an under-sampling of the glyph's surface. To remedy this issue, more advanced shading techniques need to be used. First, an image-space shading pass, based on deferred shading will remove the high frequencies from the normal vectors by performing a re-estimation of normal vectors of small particles. This will enhance the perception of the shape of the data as a whole. To further enhance the depth perception, ambient occlusion will be applied to the visualization. Both approaches are optimized to work with particle data but can also be applied to other applications.

2.4.1 Image-Space Normal Estimation

When displaying hundreds of thousands of particles using ray casting of spherical glyphs on a standard workstation display, the glyphs will often have an image-space footprint of less than a single fragment. Due to this huge under-sampling the normal vectors and thus the resulting shading will exhibit high-frequent noise obscuring the shape of the data with visual clutter (cf. Figure 23). Similar problems, known e.g. from point-based rendering, are typically addressed using pre-filtering [ZPvBG01] or by accumulating pre-filtered input for the lighting computation [BSK04], [HSRG07]. As the centre images of Figure 23 show by applying costly super-sampling, the results are not completely satisfying, even with these approaches. These images were created by non-interactive 8×8 super-sampling, i.e. rendering at resolution 8192×8192, followed by down-sampling to obtain the final image at 1024×1024 pixels. Still, the centre image of the upper row exhibits aliasing ring effects on the large particle block. Instead of filtering normal vectors, which are generated on entities not noticeable at all (individual spheres) normal vectors can be re-created for the visible information. This generates a coherent impression of the large-scale structure of the data set and automatically adapts to the viewing distance (right images in Figure 23). The method has been previously presented in [GRDE10].

In the image-space fragment shader of a deferred shading pass, a 3×3 fragment patch is read from the depth buffer, centred at the fragment to be shaded. These nine values are used to set up a quadratic Bézier patch, which captures orientation and curvature of the surface formed by the particles in the data set. The normal vector for the lighting operation is then evaluated at the centre of the patch. To maintain the correct normal vectors for large glyphs, the ray casting fragment shader can write the normal vectors into the parameter buffer including a confidence value based on the glyph image-space foot-print. Based on that value the normal vector used for shading can be interpolated between the glyph's normal

vector and the estimated normal vector, providing a smooth transition when zooming in or out.

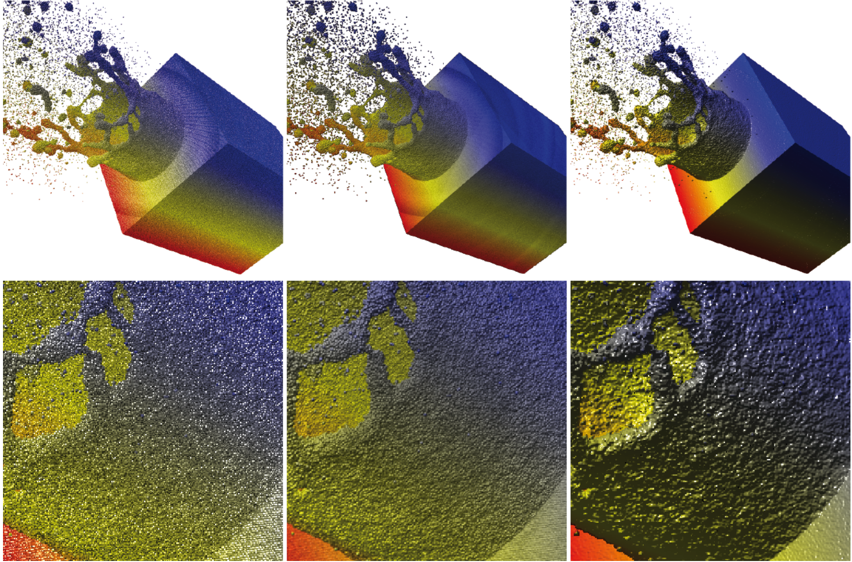

Figure 23: Comparison between local lighting and deferred shading. Left: straight-forward ray casting of spheres. Middle: 8 × 8 super-sampling of ray cast spheres. Right: deferred shading with estimated normal vectors.

2.4.2 Ambient Occlusion for Particle Data Sets

A second approach to further emphasise the global structure of a data set and to remedy the perception problem, which arises from aliasing and visual clutter if the particle-to-pixel ratio drops, is to add visual cues to the resulting image. Such cues can roughly be categorized into depth-based and lighting-based cues. Depth-based cues explicitly hint at abrupt changes of the depth, i.e. distance to the camera. The most basic method is fogging, which simply blends the colour of a pixel with a specified fog colour based on the corresponding depth value. It is easy to implement because of the trivial evaluation and it is even built in to graphics APIs. However, since this is a global image-space approach, even non-occluded particles might become hard to perceive if their distance to the camera is too large.

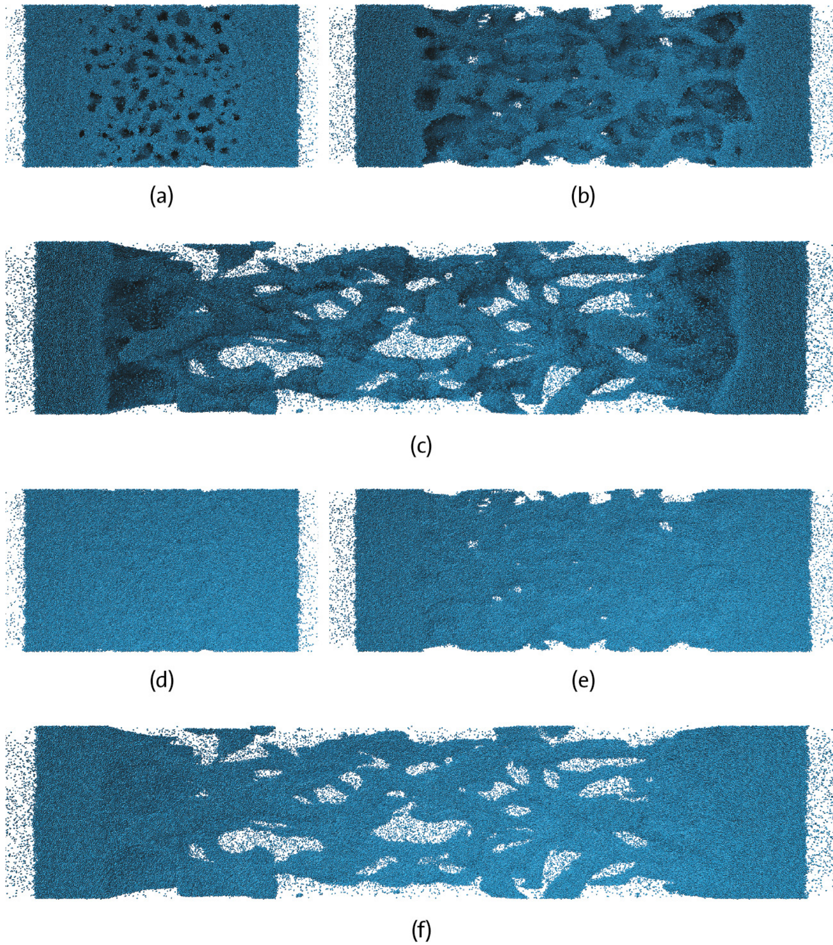

Figure 24: MD simulation of 2 million molecules forming a liquid layer of argon in vacuum, which gets ripped apart by its own vapour pressure. The time frames shown are 5 (a)(d), 15 (b)(e), and 30 (c)(f). The lower images (d)(e)(f) show straight-forward ray casting of spheres for the individual molecules. The upper images (a)(b)(c) show the same rendering enhanced with OSAO. Especially the break-up of the structure in time frames 5 and 15 is more clearly visible using OSAO.

Lighting-based cues are often generated by methods which are also used to increase realism in computer graphics and are often approximations of global illumination models. The most prominent example might be shadows, which visually connect objects with their surroundings. A user study performed by Lindemann and Ropinski [LR11] has shown that shadow effects always have positive effects on shape perception and depth perception in volume rendering, although the users' subjective opinions did opt for simpler shading. Visualizations of large particle data sets have similar perception problems as volume rendering, although caused by different rendering problems: i.e. aliasing and visual clutter. Thus enhancing the rendering of particle data by shadow-like effects approximating global illumination alleviate this problem, as can be seen in Figure 24. However, implementing shadows for particle-based data sets is challenging, because of the nature of the data. The large number of small particles results in high frequencies to be represented. Thus, there are no obvious, large, continuous structures which could be used to cast approximate shadows and the discrete resolution of shadow maps often increases the aliasing issue of the particle data. But even if these problems are correctly addressed, hard shadows have an issue similar to the one mentioned for fogging: non-occluded particles not in the vicinity of other particles might be less easily perceptible.

Object-space Ambient Occlusion (OSAO), introduced by Zhukov et al. [ZIK98], seems a fitting solution to this problem. They modelled the ambient lighting term in local lighting equations as radiating, non-absorbing and completely transparent gas equally distributed throughout the whole scene. This ambient lighting value is evaluated from the amount of this gas surrounding scene surfaces, which again depends on the distances to neighbouring scene objects, figuratively speaking, how much potentially incoming light being blocked by surrounding geometry. As OSAO is thus based on local geometry properties, which are the features to be emphasised, this approach is a suitable choice. The method of applying OSAO to large, time-dependent MD data sets for interactive rendering was first published in [GKSE12]. The core idea is to aggregate the particle information in a coarse-resolution density volume, as approximation of the scene information which can be efficiently evaluated in real-time without relevant impact on the rendering performance.

Another technique for OSAO which allows fast rendering was presented by Pharr [PG04]. His method relies on pre-computation of the occlusion factors, which makes it unfeasible for large, dynamic data sets. The same is true for the method originally developed by Sattler et al. [SSZK04] and later used by Tarini et al. [TCM06] in their molecular visualization program *QuteMol*. This method samples a set of directions for each vertex in the scene to determine the occlusion. Since it scales linearly with the number of vertices, it is not feasible for large, dynamic data sets. Bunnell [Bun05] presented an OSAO method for dynamic data, which was later refined by Hoberock and Jia [HJ07]. Proxy elements (discs), which are used for shadow approximation, are generated for each polygon. The performance is optimized by using a hierarchical clustering as coarsening for distant objects. The work of

Papaioannou [PMP10] uses volumes to store the occlusion information, similar to the approach presented in this thesis, but is also optimized for polygonal data. *Ambient occlusion volumes* by McGuire [McG10] are similar to classical shadow volumes [Cro77]. The technique computes an ambient occlusion volume for each polygon, utilizing a geometry shader. One could argue that all these polygon-based methods could be applied to particle data using a minimal tessellation for the spherical glyphs. However, to reasonably represent a sphere at least an octahedron is needed, resulting in eight triangles per sphere. For a data set of 2 million spheres (cf. Figure 24), this would result in an overall number of 16 million triangles. None of the aforementioned techniques is able to handle this amount of data interactively, as the results sections of the corresponding publications show.

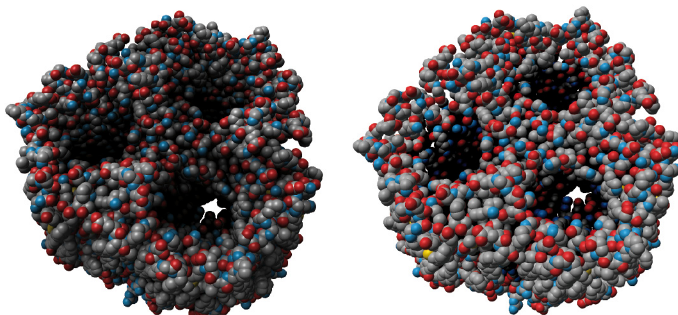

Figure 25: Maltoporin protein renderings (PDB-ID: 1AF6, 10 000 atoms); Left: the presented OSAO method; Right: depth darkening [LCD06]; Depth darkening emphasizes the three channels but is not able to extract the more shallow structures on top of the protein.

In computer graphics, approximations of ambient occlusion in image space are widely used, as their computation times only depend on the resolution of the viewport and these methods are thus suited for large and complex scenes. They have therefore evolved into a de-facto-standard for approximated global illumination in real-time applications (e.g. computer games). These methods are collected under the term *screen-space ambient occlusion* (SSAO). However, as they only take the visible parts of the scene into account, they are prone to create imperfect, often even incorrect shadowing effects. The simplest method is the so-called *depth darkening* by Luft et al. [LCD06], which essentially implements depth-dependent halos by blurring the depth buffer. While it can be computed quite fast, the achieved effect resembles actual ambient occlusion only vaguely (cf. Figure 25). Mittring presented a more advanced SSAO variant, which was integrated in the game Crysis by Crytek [Mit07], and which was later refined by Kajalin [Kaj09]. This method estimates the amount of solid geometry surrounding a point of interest by random-

ly sampling the vicinity in object-space. Subsequently the samples are projected into image space and compared to the depth buffer. The ratio of samples lying within and outside of solid geometry is used as approximation of the occlusion geometry. This technique, like all SSAO methods, can only create shadows from objects visible in the final image, which can result in erroneous shadows: e.g. a small object near the viewer can occlude and thus shadow a larger object being farther away (cf. Figure 30). Fox and Compton developed *ambient occlusion crease shading* [FC08], which enhances the boundaries of displaced objects, not unlike depth darkening, but also takes the normal vectors into consideration, like Mittring's technique. Similar to the method of Hoberock and Jia [HJ07], Shanmugam and Arikan [SA07] presented a combination of near-field occlusion and distant occlusions represented by *occluder geometry* (spheres) which approximate distant geometry. While their approach creates acceptable image quality, it is unfeasible for the large, dynamic particle data sets or in-situ visualization, since the occluder geometry cannot be precomputed. Shopf et al. [SBSO09] proposed to use deferred shading to compute the occlusion for all objects in screen space employing polygonal proxy objects. A presented example scene with 5000 spheres thus results in 60 000 triangles due to the approximation with cubes, prohibiting the usage of this method for large data sets. The image-space horizon-based ambient occlusion by Bavoil and Sainz [BS09] was included as demo in the Nvidia DirectX SDK 10 [Nvi]. The ambient occlusion is computed from the altitude angles between random samples retrieved from the depth buffer and a viewing horizon. Visible artefacts occur for lower numbers of samples, which can only partially be reduced by blurring, while high numbers of samples impede fast rendering.

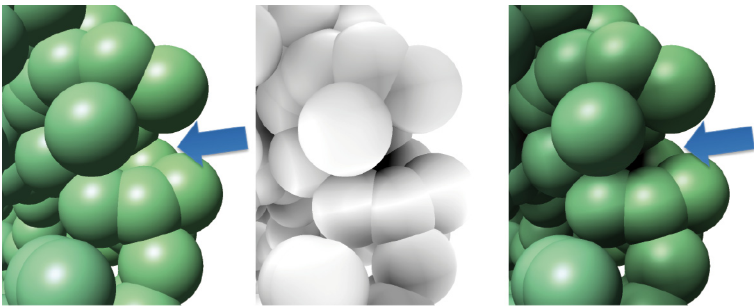

Figure 26: Zoomed-in view; left: local lighting and Phong shading only. Since the spheres are overlapping the depth structure is not clear, especially for the marked sphere. Middle: the ambient occlusion factors; right: the final image; The position of the marked sphere is now clearly behind all other spheres.

The core idea of ambient occlusion is to compute factors $A_{\boldsymbol{p}}$ for ambient light for each (visible) point $\boldsymbol{p}$ in the scene, which resembles the lack of incident secondary light rays reaching this point [ZIK98], because of nearby objects. Thus $A_{\boldsymbol{p}}$ is given by the integral of the distances to occluding geometry over the visible hemisphere hS^2, using an maximum distance to limit the calculation to a local neighbourhood. The final result is then generated by multiplying these factors with the result of local lighting (cf. Figure 26).

$$A_{\boldsymbol{p}} = \iint_{\boldsymbol{x} \in hS^2} \rho\big(L(\boldsymbol{p}, \boldsymbol{x})\big) \cos\alpha \, dx \tag{3}$$

Equation (3) is a slight variant of the original equation from Zhukov et al. [ZIK98] to calculate the ambient occlusion factor, with $\rho(L)$ being the blocked incoming light energy based on the distance of the nearest obstacle of the scene $L(\boldsymbol{p}, \boldsymbol{x})$ in direction $\boldsymbol{x}$. As α is the angle between $\boldsymbol{x}$ and the surface normal $\boldsymbol{n_p}$ at $\boldsymbol{p}$ the $\cos(\alpha)$ is a basic form factor for the incoming light and can be rewritten using a dot-product. For mesh-based geometry $\bar{p}$ is selected to represent a patch (triangle or quad) and the resulting $A_{\boldsymbol{p}}$ is interpolated between the patches to get a continuous result:

$$\mathrm{A}_{\mathbf{p}} = \sum_{\overline{p_\iota} \in P_{\boldsymbol{p}}} \omega(\boldsymbol{p}, \overline{p_\iota}) A_{\overline{p_\iota}} \tag{4}$$

with $P_{\boldsymbol{p}}$ being the patches in proximity of $\boldsymbol{p}$ and $\omega(\boldsymbol{p}, \overline{p_\iota})$ is the interpolation weight factor for the value of the patch $\overline{p_\iota}$ when evaluating at position $\boldsymbol{p}$, which can simply be bi-linear or using barycentric coordinates.

This approach is not directly applicable to particle data sets for two reasons: First, the definition of the patches $\overline{p_\iota}$ is not simple, since a single particle is depicted as sphere and would thus require multiple differently-oriented patches. Second, the integration over the hemisphere or the summation over the neighbourhood is usually implemented by rendering into FBOs. The pure number of required evaluations would hinder interactive rendering, and, even worse, the discrete resolution of the FBOs would have to be very high to capture the intrinsic high frequency of particle data sets.

However, equation (4) is applicable because $A_{\boldsymbol{p}}$ is a continuous function over $\boldsymbol{p}$, which is intuitively clear as the integral in equation (3) basically represents the local neighbourhood of $\boldsymbol{p}$ which will change only little if $\boldsymbol{p}$ changes only little. $A_{\boldsymbol{p}}$, therefore, does not contain high frequencies, even though the original particle data does, as these frequencies are *smoothed out* by the integral. Thus, the need to represent these high frequencies in the first place is not given at all; meaning $\rho(L)$ can be approximated by a smooth representation of the particle neighbourhood. The blocking of light rays can be related to the density of the particles. Equation (3) can thus be re-written to:

$$A_{\boldsymbol{p}} = \iint_{\boldsymbol{x} \in hS^2} D(\boldsymbol{p} + \lambda \boldsymbol{x}) \boldsymbol{n}_{\boldsymbol{p}} \cdot \boldsymbol{x} \, dx \tag{5}$$

with D being a density values calculated from the particles and λ being a sampling distance, which corresponds to the range for the contributing neighbourhood. Because of this the particle data can be represented as coarse-resolution density volume. Using only one value, at distance λ from $\boldsymbol{p}$, a single volume cell is sufficient to deliver all data of the relevant neighbourhood in direction $\boldsymbol{x}$. Thus, the volume resolution defines the visible neighbourhood and λ is half the length of a voxel. This directly corresponds to L_{max} in [ZIK98]. Because of these assumptions, to take a reasonable neighbourhood into account, the resolution of the density volume has to be rather coarse, which is even beneficial for the rendering performance. Additionally, the tri-linear interpolation, built into GPUs, provides a simple solution to ensure the required smoothness of the data. The coarse resolution of the density volume also results in the fact that the integral over the hemisphere in equation (5) will only fetch very few different data values as λ scales the unit hemisphere to have a diameter of exactly one voxel. Furthermore, the form factor $\boldsymbol{n}_{\boldsymbol{p}} \cdot \boldsymbol{x}$ will reduce the influence of values fetched at the areas of the hemisphere close to the surface patch's tangent plane. Thus we can simplify the approximate occlusion geometry value to a single texture fetch of the most relevant, interpolated density value per patch:

$$A_{\boldsymbol{p}} \approx D(\boldsymbol{p} + \lambda \boldsymbol{x})$$

This is only possible because of the nature of the particle data sets as described above. But to some extent it can also be applied to other types of data, as long as a smooth representation like the coarse-resolution density volume is possible and reasonable.

There are several possible implementations to generate the particle density volume. Following the scattering approach, a representative, in this case a solid sphere, is splatted into the correct volume cell:

$$V = \mathrm{g} \sum_{\boldsymbol{p} \in V_{bounds}} \frac{4}{3} \pi r_{\boldsymbol{p}}^3 \frac{1}{V_{size}^3} \tag{6}$$

with V being a voxel value, $\boldsymbol{p} \in V_{bounds}$ all particles contributing this voxel, $r_{\boldsymbol{p}}$ the radius of the particle, and V_{size} being the length of the edge of the voxel volume. The factor g is explained below. Equation (6) has three issues:

First, the density might be overestimated. Consider the worst case of two spheres of the same size at the same position resulting in a summed-up volume twice as large as would be correct, assuming a maximum value of density of one for full opaqueness. This, however, is not relevant for data from molecular dynamics. When visualization and simulation are parameterized correctly, i.e. the sphere

radius corresponds to a significant radius of the simulation force field (e.g. the Lennard-Jones radius) the spheres may only overlap very slightly.

As second issue we have to keep the closest packing of non-overlapping spheres in mind if deriving opaqueness from the density of particles. If correctly modelled, a summed-up sphere volume of $\approx 74\%$ of the voxels volume must be considered a completely filled voxel. But to generate a completely opaque voxel far less spheres are sufficient. Observed from one direction, two perpendicular layers of spheres, similar to the alignment in close packing, are sufficient to completely block any light rays in this direction. Obviously, this cannot be addressed in general by just one density value. Thus the factor g is introduced to adjust the density-to-opaqueness conversion. Empiric studies, however, showed that even $g = 1$ results in good image quality and that tweaking the parameter is not necessary at all.

The third issue of equation (6) stems from the fact that $\boldsymbol{p} \in V_{bounds}$ does not explain what happens with spheres being only partially inside the voxels' bounds. A simple solution is to virtually move the sphere slightly to be fully inside a single voxel, by choosing the voxel the sphere's centre lies inside. This way, each sphere contributes to only one voxel and always contributes with its full value. Since the volume resolution has to be coarse, this virtual displacement is negligible.

This splatting method can be implemented in various ways. One of the simplest is a multi-threaded CPU implementation. The volume texture is generated at scene setup with the chosen resolution, as well as a corresponding linear float-array representation in main memory. For each frame the float array is cleared to zero and completely rebuilt. A simple parallelization with OpenMP [Ope08] allows to utilise multi-core CPUs. To avoid explicit synchronization the calculation is split into two passes. First, a parallel loop over all particles computes the volume values based on equation (6). Each thread writes into a float array of its own. The second pass then sums these arrays up into the final array to be uploaded to the GPU, parallelized over all voxels.

Obviously, the question arises if the volume can be generated directly on the GPU. A classical rendering approach would be a binning using a GLSL shader into a texture. The volume is bound slice-by-slice, e.g. along the z axis, as render target and the particles contributing to the bound slice are rendered. The particles are rendered for each slice and the GLSL shader decides whether to generate a fragment on one of the texels/voxels or to discard the particle for this slice. The particles can also be sorted beforehand allowing to only render particles which are likely to contribute to the currently active slice. Utilizing vertex arrays and index arrays allows for fast data transfer in this case. Obviously, if the particle data will be used more often and if it fits into the GPU memory, using VBOs is even more favourable. The fragment shader can simply output the value of the particle's sphere volume and the summation of equation (6) is realized by additive blending.

However, the GPGPU API CUDA does offer a more flexible way to utilize the graphics hardware. The *Particles* demo from the Nvidia GPU Computing SDK [Gre08] shows a good solution for an implementation of particle binning, as it

also utilizes a spatial data structure. The particles are binned into a grid by means of sorting based on a spatial hashing: Each cell of a uniform grid has a unique hash value according to its position. A first CUDA kernel computes the hash value for each particle, i.e. the grid cell the particle's centre point lies in. The particles are sorted based on these hash values using a parallel radix sort. A subsequent compute kernel collects the start and end indices for each grid cell finally resulting in the number of particles and their volume contributions, in case of differently-sized particles.

Different from Nvidia's *Particles* demo, in which the particle positions were uploaded only once at the simulation start, for the visualization of dynamic MD data, the positions have to be transferred to the GPU for every frame. As long as the size of the data allows, using VBOs will be beneficial, because the data is at least required twice, first for the splatting CUDA kernels and then for the final rendering. As CUDA is capable of directly accessing VBOs (*cudaGLMapBufferObject*), storing the data in graphics memory is very efficient.

After generating the particle density volume with one of the presented approaches the final image can be rendered. However, there are several possibilities for the evaluation of the ambient occlusion factor A_p for each particle. Following the original idea, we can subdivide each sphere glyph of each particle into several patches. Inspired by environment mapping we subdivided each sphere into six patches oriented in directions of the positive and negative world-space main axes. In the vertex shader these six values from the particle density volume are fetched (at distance λ along all main axes directions from the particle's position) and then transferred to the fragment shader.

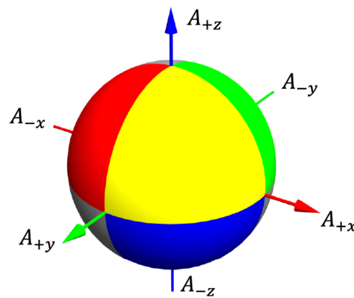

Figure 27: Interpolation of the ambient occlusion factors is based on the surface normal. The values along the main axes (arrows) are interpolated with the squares of the components of the normal vector. In the general case, all six values are thus required.

The fragment shader interpolates between these six values utilizing the already calculated surface normal vector $\boldsymbol{n} = (n_x\ n_y\ n_z)^T$. Based on the signs of the components of the vector three of the six values $(A_{\pm x}, A_{\pm y}, A_{\pm z})$ are chosen (A_x, A_y, A_z) to be interpolated. In the resulting spherical triangle the interpolation is performed by using the square of the normal vector components as barycentric coordinates: $A_f = n_x^2 A_x + n_y^2 A_y + n_z^2 A_z$. In the general case, all six values transferred from the vertex shader will be required for shading a sphere as can be seen in Figure 27.

A second approach would be to subdivide the spherical glyph further into patches, such that each resulting fragment is a patch of its own. The fragment shader can then fetch the value from the particle density volume itself from a position p given by the particles position and the normal vector: $A_f = D(\boldsymbol{p} + \lambda \boldsymbol{n})$. Apart from the tri-linear interpolation taking place by this texture fetch, no further interpolation is required.

Both approaches can be compared by the number of required texture fetches, which still are one of the more costly operations on GPUs. Obviously, the break-even point is reached when a sphere reaches a screen-space footprint of six fragments. When rendering large number of particles, this is likely to be the case. This approach to generate OSAO for particle data has been evaluated with six differently-sized data sets from real-world applications (cf. Table 9).

Table 9: Data sets used to test the presented OSAO method.

Data set (Figure)	# Particles	Volume resolution
1AF6 (Figure 25)	10,000	$8 \times 8 \times 8$
NiAl (Figure 31)	36,500	$32 \times 32 \times 32$
CCMV (Figure 29)	220,000	$18 \times 18 \times 18$
Laser cross (Figure 30)	560,000	$16 \times 16 \times 16$
Exp2mill (Figure 24)	2,000,000	$16 \times 128 \times 16$
Laser big (Figure 32)	11,800,000	$128 \times 32 \times 32$

The rendering performance of all presented methods was evaluated and is shown in Table 10. The test system was an Intel Core i7 x980 (6×3.3 GHz) with 12 GB RAM and an Nvidia GeForce GTX 580 (1.5 GB VRAM). The viewport resolution was set to 1280×720 pixels for all measurements and the data sets were maximally zoomed while still being entirely visible.

Table 10: The performance results of the presented OSAO method, including generation of the particle density volume and rendering. All values are in *frames per second*. The rendering mode *no AO* refers to a renderer without any ambient occlusion calculations (i.e. no density volume is generated). The rendering mode *none* describes the test without rendering where only the density volume is computed and transferred to the graphics card.

Data set	Rendering mode	CPU	GLSL	Cuda
1AF6	No AO	716	-	-
	Vertex	630	559	422
	Fragment	626	563	417
	None	2140	983	1000
NiAl	No AO	821	-	-
	Vertex	786	379	492
	Fragment	797	373	488
	None	981	411	759
CCMV	No AO	254	-	-
	Vertex	212	72.8	147
	Fragment	214	72.7	152
	None	700	91.8	430
Laser cross	No AO	114	-	-
	Vertex	104	33.0	71.8
	Fragment	104	33.3	71.8
	None	340	37.6	221
Exp2mill	No AO	72.8	-	-
	Vertex	57.8	9.91	41.9
	Fragment	59.9	10.0	41.9
	None	144	10.6	73.0
Laser big	No AO	14.3	-	-
	Vertex	7.43	1.10	6.15
	Fragment	7.67	1.11	6.30
	None	27.0	1.36	14.11

All implementations of the particle density volume generation are able to maintain interactive frame rates for data sets with up to two million particles. The GLSL implementation, however, has much lower performance, due to the slice-based rendering, which introduces high latencies, and requires many costly changes of the OpenGL state. CUDA performs much better in this regard, but due to the small sizes of the density volumes the small problem size seems not to be able to fully utilize the GPU and to compensate for the additional overhead CUDA introduces. While the number of particles and the number of cells (especially for the Laser big data set) seem sufficient for the parallelisation of the stream processors,

the calculations are most likely to simple and thus to fast to fully harvest the GPU power. In comparison the multi-core CPU, with its 12 local cores, seems to be able to adapt better and to perform better scheduling for this task. The CUDA implementation is therefore, although reaching interactive frame rates, significantly slower that the CPU implementation.

The ambient occlusion factor retrieval methods have only small impact on the frame rate. For larger data sets it is to be expected that each particle will cover less than six fragments on the screen, resulting in the fragment-shader based method to perform better. The performance measurements confirm this expectation to be true. However, the difference in performance is not an issue, since it only occurs for small data sets (below 1 million particles), which can be rendered at very high frame rates anyway. Comparing the image quality, the fragment-shader-based method yields better-looking results (cf. Figure 28).

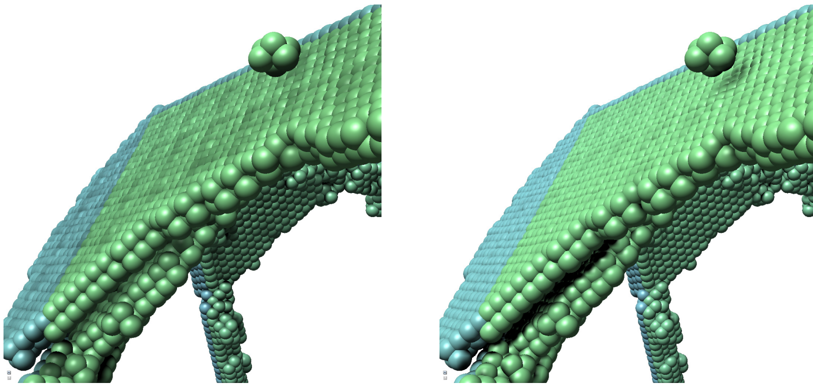

Figure 28: Comparison of image quality of the vertex-shader-based OSAO evaluation (left) and the fragment-shader-based evaluation (right). The fragment-shader-based approach better captures the neighbourhood information, as can be seen by the shadow of the small floating cluster at the top right, and the dark gap between the different atom layers at the bottom left.

Figure 29 and Figure 30 show image quality comparisons for two exemplary data sets: the CCMV virus and the laser cross laser ablation MD data set. In both figures the upper left image shows naïve ray casting with normal phong lighting only. The upper right images were generated by VMD [HDS96] using the Tachyon [Sto98] ray tracer which also computes ambient occlusion. These images serve as ground truth. Unfortunately, it was not possible to deactivate the drop-shadows, which are especially visible through the expelled particles in Figure 30. The lower left images show the results from SSAO [Kaj09]. The lower right images are generated with the presented OSAO method. All parameters for all methods were adjust-

ed to produce the best possible visual results, by reaching a trade-off between shadowing effects and visible artefacts.

For the CCMV virus data set in Figure 29 the naïve ray casting fails to show the substructure of the surface. The structure can be seen employing any of the other three methods. While the SSAO shows the structure only barely, ray tracing and OSAO clearly show the circular pits. However, both methods result in slight *over-shadowing* effects. But the shape of the surface remains clearly visible.

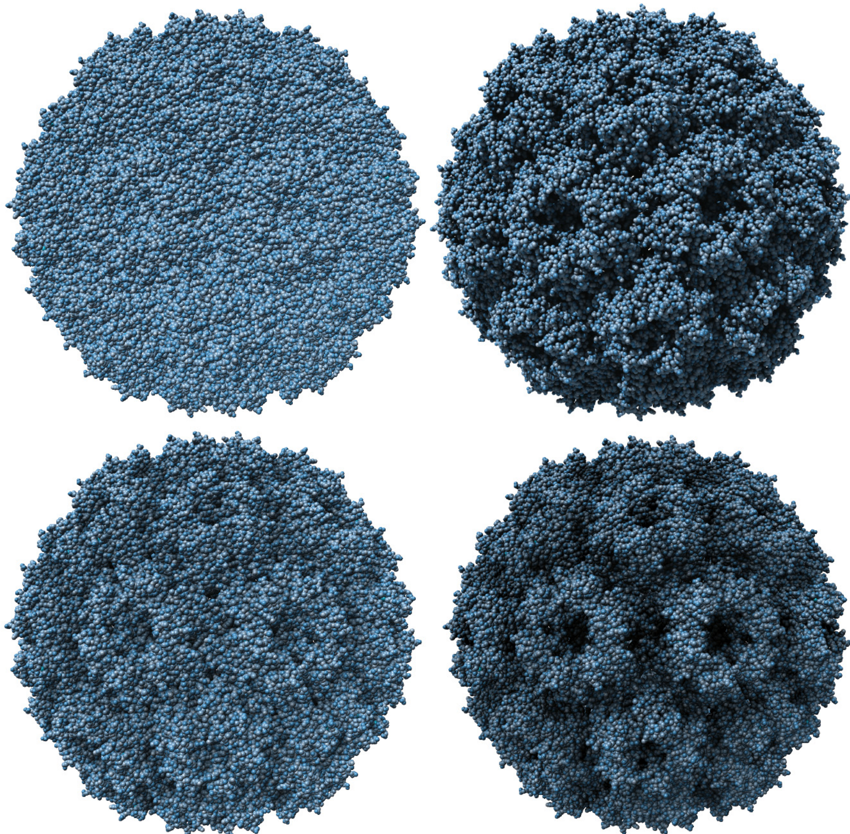

Figure 29: Comparison of different Shading techniques: top-left: naïve ray casting, top-right: ray tracing with ambient occlusion; bottom-left: SSAO; bottom-right: the presented OSAO approach. Data set: Cowpea chlorotic mottle virus with 220,000 atoms.

The laser cross data set in Figure 30 clearly shows the drawback of SSAO methods. The expelled atoms cast rather big, blurry shadows onto the block of material while failing to emphasise the depth of the cleft cut by the laser. The result of OSAO is very close to ray tracing.

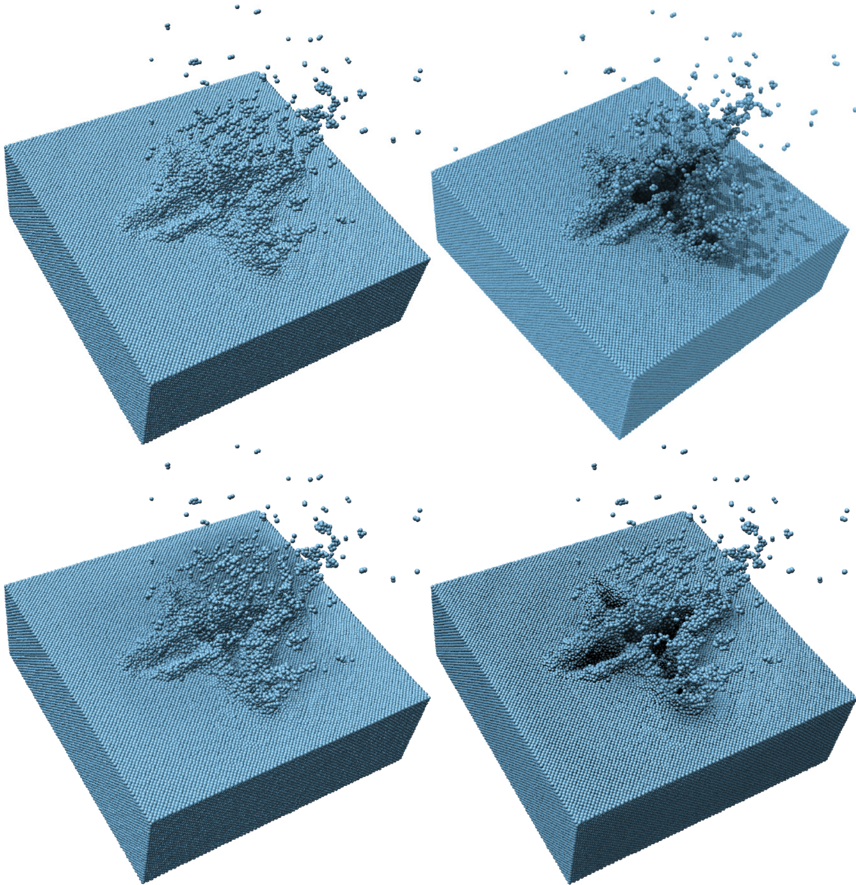

Figure 30: Comparison of different Shading techniques: top-left: naïve ray casting, top-right: ray tracing with ambient occlusion; bottom-left: SSAO; bottom-right: the presented OSAO approach. Data set: laser ablation with 560,000 particles.

However, due to the simplifications of the presented OSAO technique, this method can generate visual artefacts in special cases. The data set shown in Figure 31 contains planes formed by the particles which are not aligned with the world-

space main axes. Either enhancing the particle density volume with gradient information or optimising the AO factor sampling could be used to remedy this issue. But, as can be seen in the two left zoomed-in views, the artefacts are barely visible when using normal parameter settings, while other structures of the data set, like the gap between the two atom planes (upper view) or the hovering atom cluster (lower view) are nicely emphasised.

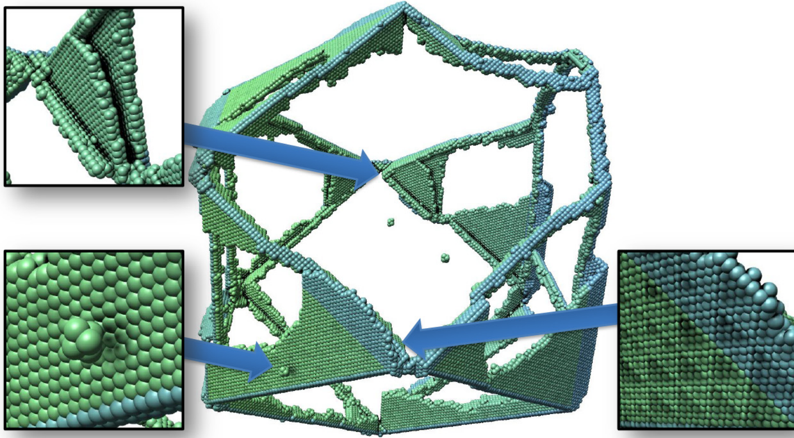

Figure 31: The nickel-alumina data set (NiAl) exposes visual artefacts. While the overall structure is clearly emphasized (cf. zoomed-in views on the left), the flat planes formed by the crystal lattice are not aligned with the particle density volume. The right zoomed-in view shows the artefacts with exaggerated density volume influence factor g to make them better perceptible.

Figure 32 shows a large data set from a laser ablation simulation consisting of over 11 million atoms. The structure of the crater's brim is not emphasised as it is on a too large scale compared to the used density volume resolution, which does define the visible neighbourhood for the presented OSAO method. A solution to address this issue would be to decouple the neighbourhood evaluation from the grid resolution, either by splatting smoothed kernels, filtering the density volume or by stochastic scattering to make long-range neighbourhood information available. In addition, a more complex value retrieval method would then be beneficial, e.g. ray marching or utilizing directional smoothed occlusion volumes. This would require larger particle density volumes and more sophisticated techniques and construction algorithms like [KS09]. However, these extensions are orthogonal to the goal of the presented OSAO, which was designed to be as simple as possible and to achieve the best result possible at the same time.

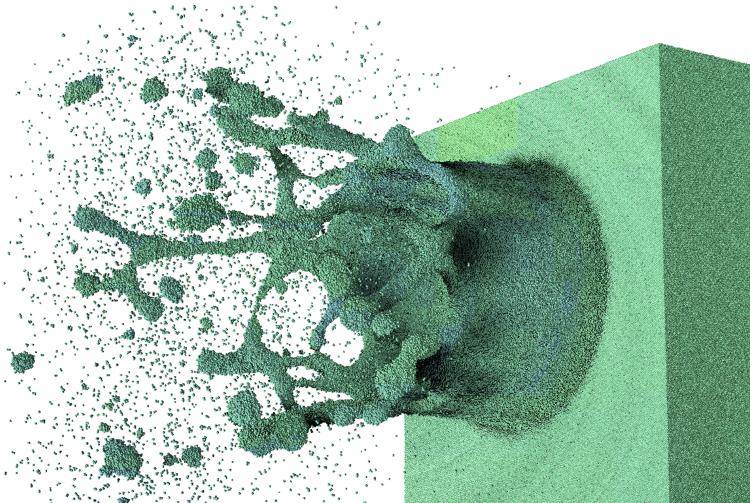

Figure 32: A laser ablation data set of 11.8 million particles; the shape of the crater's brim is not emphasized because the OSAO only uses short-range neighbourhood information to determine the occlusion factors.

The presented OSAO is a good solution for most MD data sets, as it only produces visual artefacts in very special cases, is easy to implement, and yields very good image quality in the general case. As long as splatting into the particle density volume can be applied and the retrieval of the ambient occlusion factors is adapted, the method is applicable to other types of data as well. In the context of this thesis this is especially interesting for particles not being represented by simple spheres, but by more complex glyphs.

2.5 Complex and Composed Glyphs

While spheres are often sufficient to represent atoms in the context of MD simulations from thermodynamics, more complex glyphs are required to reasonably represent molecules with non-spherical mass distribution or elements with distinct orientations like electrostatic or magnetic dipole moments. The ray casting method is even then favourable because of the superior visual quality. Using iterative approaches, even arbitrarily shaped, curved surfaces can be generated at high performance (e.g. see [KHK+09] or [SGS06]). This, however, would result in a piece-wise description of the glyph, requiring multiple rendering calls. Up to a certain complexity glyphs can be directly ray cast with a single, non-iterative shader following the method presented in chapter 2.1. Reina and Ertl [RE05] showed how to extend the

method to dipole glyphs (cf. Figure 33) for Lennard-Jones representations of molecules with two mass centres and correspondingly-aligned dipole moment. In their work, the mass centres are represented by two spheres and the dipole moment is visualized using the metaphor of a bar magnet using a bi-coloured cylinder.

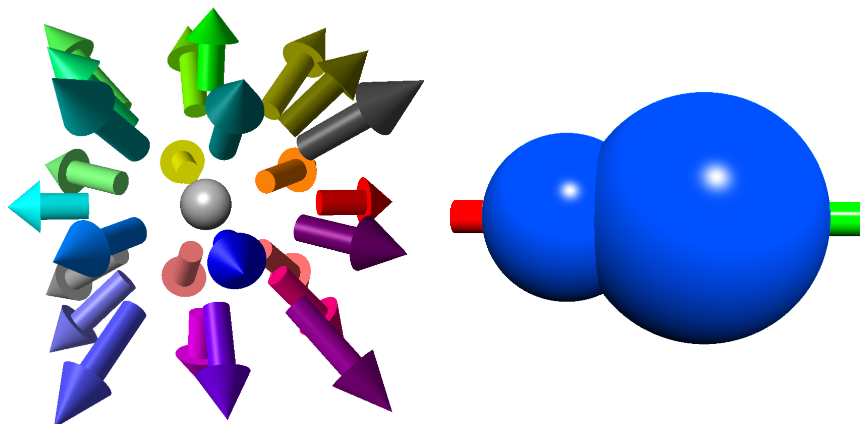

Figure 33: Simple compound glyphs, which can be ray cast in a single optimized GPU shader; right: arrow glyphs consisting of cylinders and cones, colour-coded based on their direction; left: dipole-glyph consisting of two spheres representing mass centers and one two-coloured cylinder representing a magnetic dipole moment [RE05].

The principal idea is the same as already described: ray casting is performed in a defined glyph space, in which the glyph has a simple orientation and is placed at the origin. The estimate of the image-space footprint is performed by the described forward projection of the object-space bounding box of the glyph. The central aspect of the ray casting of the dipole glyph is now to ray cast all three elements, two spheres and one cylinder, simultaneously inside the same glyph space, and choosing the surface hit closest to the viewer, including the corresponding surface colour and normal vector.

This concept can be applied to other glyphs constructed from simple shapes, which are reasonably aligned in a common glyph space. A good second example is the arrow glyph shown in the left image of Figure 33. It can be used to depict orientations, e.g. electrostatic dipole moments. The arrow is constructed from a cone and a cylinder. The common rotation axis of both shapes is aligned with the x axis in the glyph coordinate system and the tip of the cone is placed at the origin. The bounding geometry for the image-space size estimate is constructed of nine points: the tip of the cone and two times four points, each forming a square at the end of the cone and of the cylinder.

2.5.1 Glyphs Composed from Quadric Surfaces

However, for more complex glyphs, like glyphs constructed from many basic elements or constructed from elements which cannot be reasonably aligned in one common glyph-space coordinate system, the approach of using a single ray casting shader is not sensible. One example source for such shapes are thermodynamics MD simulations of ridged molecules which require multiple mass centres and multiple charges (cf. Figure 34). As the orientations of such particles to each other is relevant for their interaction and thus for the analysis process, a corresponding glyph for visualization must capture these elements. While these glyphs can also be constructed from simple shapes, like spheres and cylinders, the sheer amount of graphical primitives poses a significant challenge for interactive visualization, as the number of graphical primitives, even for rather simple glyphs, is likely to be one order of magnitude larger than the number of particles. A concept of rendering such particles as well as a detailed performance evaluation was previously published in [GRE09]. This chapter summarizes the fundamental methods.

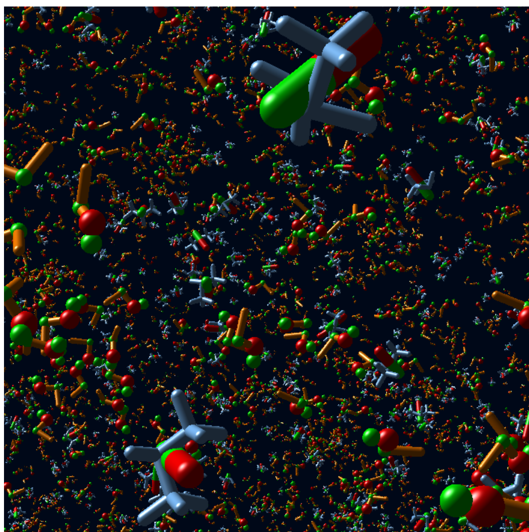

Figure 34: A typical real-world particle dataset from the field of molecular dynamics, containing a mixture of ethanol and R227ea, 1M molecules altogether, represented by GPU-based glyphs composited from 8.25M graphical primitives.

Designing reasonable compound glyphs as visual representations for such applications is not straight forward. On the one hand, the glyph must match the relevant properties of the simulation model, while on the other hand, too complex

glyphs will overload the resulting visualization and will not be useful in the end. E.g. representing the mass centres by corresponding spheres can result in large and strongly overlapping spheres, occluding and thus obscuring the actual shape of the molecule. A more sparse representation, borrowed from the visual metaphors applied to data sets in biochemistry, would be a stick representation, where the mass centres are shown by rather small spheres interconnected with their neighbours by equally-sized cylinders. This representation is sparse enough to allow placement of other elements, like charges, in the same context. To more easily distinguish molecule types, all uncharged elements in one molecule can share the same colour. Additional spheres indicate point charges, and two spheres with two cylinders show a directed charge using the metaphor of a bar magnet (cf. [Rei08]). The radii of these elements are chosen proportionally to the strength of the charges they represent and the sign is shown by the colour (green for positive and red for negative charges).

Figure 35: Two complex molecules modelled with spheres and cylinders. Left: an ethanol molecule with the orange stick representing the carbon backbone and three spheres showing point charges; Right: an R227ea molecule with a blue stick representation of the carbon and fluorine atoms and a bar magnet showing a directed charge.

Figure 35 shows two molecules used in thermodynamics MD simulation modelled using these principles. The ethanol molecule (left image) consists of four spheres and two cylinders (a third cylinder, located between the two upper spheres, is automatically removed in an optimization step because it is completely occluded). The R227ea molecule (cooling agent heptafluoropropane, right image) is constructed from twelve spheres and eleven cylinders. Of course, other representations for these molecules could also be applied. Similar to the approach for the simpler compound glyphs, the shape primitives are placed and oriented in a particle-centred coordinate system. Since the required graphical primitives, i.e. spheres and cylinders, can already be ray cast, the naïve approach would be to transfer all these primitives from their local, glyph-centric coordinate systems to a common world-space coordinate system. This can either be carried out on the CPU to keep the ray casting shaders as simple as possible, or this can be calculated by the GPU, requiring some additional attributes per glyph. Especially when looking at possibilities to generate

the graphical primitives directly on the GPU from the original particle data to minimize the data transfer load, the latter option seems beneficial.

The mechanism of hardware-supported *instancing* [Ins] is a possible choice for this approach. For hardware not supporting this function, nowadays only rather old graphics cards, instancing can be emulated using VBOs (cf. [PF05]). The idea is to upload all particle data once per frame and re-use it once per graphical primitive needed for the specific molecule glyph. Based on the instancing index (*gl_InstanceID*), or the corresponding uniform shader variable, the parameters for the primitive to be generated for each glyph (such as relative coordinates, radius, and colour) can be loaded (e.g. from uniform values, or fetched from a texture) into the vertex shader and passed on to an adjusted ray casting code. There, these relative coordinates are combined with the glyph position and orientation to obtain the world-space coordinates for the graphical primitive. Since the particle data is, at least temporarily, stored in graphics memory, the data transfer load is minimized and the rendering performance scales well.

Another technique to reach the goal of creating compound glyphs would be the geometry shader. The particle data can be uploaded as point primitives, similar to ray casting of simple spheres. For each molecule a geometry shader can be compiled, which outputs the point primitives for the individual graphical elements required. The parameters for these elements can again be retrieved from textures. Either a single fragment shader is required which is capable to ray cast all graphical primitives required or one geometry shader is generated for each graphical primitive. The first option would require multiple concatenated ray casting shaders and program control flow, which is expensive in graphics hardware. The latter option is clearly beneficial from a fragment-shader point-of-view, but requires again multiple draw calls, as were used for the solution based on the instancing technique. This approach was used by Lampe et al. [LVRH07] for visualising large proteins by rendering amino-acids without inner degrees of freedom.

Both solutions, using instancing or a geometry-shader, have their limitations. Instancing is bound by graphics memory to store the VBOs of the particle positions or requires multiple data upload because of the required multiple draw calls (especially when using VBO emulation of hardware supported instancing). The geometry-shader approach has a high calculation overhead and the geometry shader stage often has poor performance. However, both are viable approaches and can result in good performance, as can be seen in the original publication [GRE09].

2.5.2 Polyhedral Glyphs

Ray casting glyphs is obviously a fitting solution for particle data sets which require smooth surfaces, which can be described by compact implicit surface equations. Some data, however, require different types of particles. One example is the modelling and simulation of granular or porous media. Classical models use spherical

particles. But, to achieve realistic results comparable to laboratory experiments, data sets with 10^8 polyhedral-shaped particles are required, allowing for modelling of fine tunnel networks. Since contact points and exposed surfaces play important roles for the analysis a visualization needs to capture the particle shapes precisely. When considering polyhedral geometry, the classical mesh-based rendering looks like a viable option, since it fits the modelled shapes perfectly and there is no issue of visual quality. Glyph-based methods may, however, be beneficial because of the large numbers of particles required. Thus point-based ray casting and different implementations of geometry generation were studied to find the best-performing solution. The original publication of this study was [GRZ+10]. The data sets are constructed by stochastic distribution of rotated and scaled instances of crystallites templates as described in chapter 1.1.2. The crystallites are defined by several tangent planes on concentric spheres, each described by the single vector (cf. Figure 36). This approach results in convex objects built with convex polygonal faces as the structure can also be obtained by employing Voronoi diagrams on spheres [NLC02]. Only a few hundred such crystallites templates are used per data set.

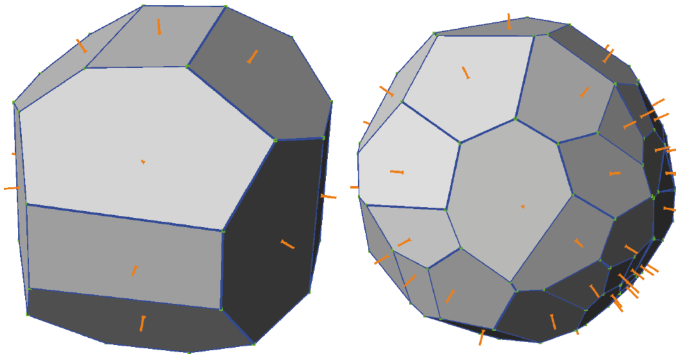

Figure 36: Two crystallites used to model the porous media; Left: rather uniform crystallite with 18 faces; Right: crystallite with 50 randomly placed faces used for performance measurements only. Orange lines show the face normal vectors defining the tangent planes.

Considering the classical polygon-based rendering, although the triangulation of the crystallites is straightforward, it is not clear whether this approach will scale to large data sets. On the one hand, a data set of 10^6 particles will easily require up to 10^8 triangles (cf. Table 11). On the other hand, the limited number of crystallite templates allows using instancing approaches, as described earlier for the composed glyphs (cf. chapter 2.5.1). Thus, four different rendering approaches have been studied for this application: using VBOs, hardware-supported instancing, mesh generation through a geometry shader, and point-based ray casting. All approaches share some common ideas: a particle-space coordinate system for each

particle in which the centre of the used crystallite is placed at the origin and in which the crystallite has a defined orientation.

Table 11: Number of triangles and vertices per crystallite for a given number of faces; the numbers vary slightly for the different crystallite templates due to the different plane cutting conditions. The two numbers of vertices show the number of *unique* vertices, required when storing the mesh data in VBOs, and the number of vertices needed to be *drawn* based on the number of triangles (relevant for the geometry shader approach).

#Faces	#Triangles	#Vertices (unique/drawn)
4	4	4/12
10	~26	~15/~46
20	~68	~36/~108
50	~200	~98/~300

The *VBO-based rendering* stores all particle crystallite templates in graphics memory. For each template the triangle mesh is computed in particle-space and stored within a VBO (using the mode GL_STATIC_DRAW for best performance). For each particle the corresponding VBOs (vertices, normal vectors, etc.) are activated and drawn. This requires one draw call per particle and thus results in a very high CPU load. The transformation from particle-space into object-space can be implemented either using the built-in model-view matrix, or by using a simple shader program.

The high CPU load can be reduced by using *hardware-supported instancing*. The only difference to the purely VBO-based approach is that the placement, scaling, and orientation of the particles are transferred into GPU memory as texture. One draw call can then draw all instances of one crystallite template at once, using the instance id to access the texels (2 texels are required when using RGBA texture; 3 float values of the positions, 1 for a scaling radius, and 4 for an orientation quaternion) holding the corresponding information for each particle. This approach reduces the CPU load to a single draw call per crystallite template. This introduces one limitation: the maximum texture sized (currently $16k \times 16k$), yielding a maximum number of particles of about 134 million per draw call. However, this limitation is easily overcome by using multiple draw calls.

Using a *geometry shader* allows to upload the particle data as point-based vertex data again. Each crystallite template is no longer stored as VBO but compiled into a geometry shader, similar to the approach of [LVRH07]. The input primitive is a single point holding the instancing parameters as vertex attributes. The output geometry is the triangle mesh (drawn as GL_TRIANGLE_STRIPs per polygon face) of the crystallite in object space. The crystallite's mesh is calculated on the CPU for the automatically generated geometry shader. Since the number of output primitives for each geometry shader is constant, the GPU should be able to perform good load-balancing to reach good performance. However, there is a maximum number of

vertex attributes a geometry shader can emit (available through the constant GL_MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS; currently 1024; cf. [Geo]). This limit results in a maximum number of 128 vertices (because multiple vertex attributes are required), yielding a maximum number of 23-24 faces per crystallite template (cf. Table 11). As the real-world data sets use crystallite templates with 18 faces, this limit is not critical, and to further overcome it, the output could be split into multiple shaders, similar to the multiple draw calls in the instancing-based rendering method to overcome the texture-size limitation. For the performance results measurements this limit was surpassed for the 50-face polyhedra, which is why these values are not available.

Finally, *point-based ray casting* can also be used to generate the visualization. The central idea is to ray cast the flat tangent planes of the polyhedron in parallel and then to choose the right hit point, similar to the composed glyphs (cf. chapter 2.5.1). For each hit point, a simple dot product of the viewing ray and the plane normal specifies whether the plane was hit front-side. The correct intersection is the hit of a front face farthest from the view point. However, if the nearest hit of a back face is in front of this one, the crystallite is not hit at all (cf. Figure 37). This approach is basically similar to the work on bounding objects for ray tracing, e.g. presented by Kay and Kajiya [KK86]. All particles of each crystallite type are now drawn with a single draw call and the corresponding ray casting shader activated, similar to the geometry-shader-based approach.

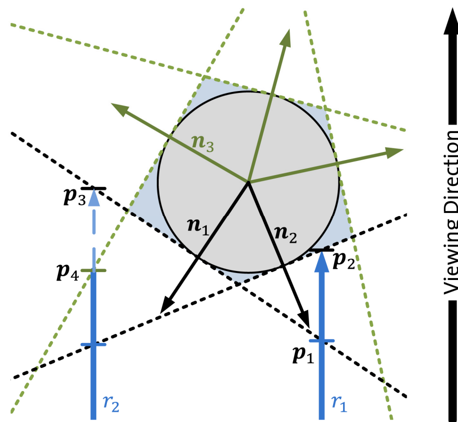

Figure 37: The principle of ray casting convex polyhedron glyphs (in 2D convex polygons). Viewing ray r_1 hits tangent plane of normal $\boldsymbol{n}_1$ at point $\boldsymbol{p}_1$. Hit point $\boldsymbol{p}_2$ is the farthest front-face hit point and thus the correct point. Viewing ray r_2 would choose $\boldsymbol{p}_3$ this way. However, the front-most back-face hit point $\boldsymbol{p}_4$ with plane of normal $\boldsymbol{n}_3$ is closer to the viewer. Thus viewing ray r_2 does not hit the polyhedron (polygon) at all.

To test the performance of the different rendering approaches, different crystallite template complexities (4, 10, 20, and 50 faces) and different data set sizes were used (statistical random distribution with 10^4 to 10^8 particles; cf. Figure 39). The measurements were conducted on an Nvidia GeForce GTX 480. The host machine was an Intel Core I7 980X 3.33 GHz, with 12 GB RAM, running Windows 7 (x64). The viewport resolution was 1024×1024. The results are shown in Table 12 and Figure 38.

Table 12: Rendering performance values in fps of the different rendering techniques; *#Faces* are the number of crystallite faces (not triangles). The rendering techniques are: VBO – VBO-based rendering, Inst – hardware-supported instancing, GPUGeom – geometry-shader based, and Raycast – point-based ray casting

#Faces	Technique	#Particles				
		10K	100K	1M	10M	100M
4	VBO	644.5	80	7.73	0.778	0.061
	Inst	1259.0	295.6	34.3	3.03	0.254
	GPUGeom	1430.1	524.8	69.0	7.51	0.747
	Raycast	181.3	81.7	25.9	4.31	0.566
10	VBO	635.5	80.1	7.7	0.758	0.064
	Inst	962.6	209.4	24.8	1.98	0.16
	GPUGeom	612.8	88.1	9.1	0.911	0.086
	Raycast	160.0	62.9	19.3	3.19	0.39
20	VBO	610.6	76.6	7.64	0.682	0.066
	Inst	592.3	105.8	11.5	1.12	0.094
	GPUGeom	174.1	19.2	1.86	0.183	0.018
	Raycast	133.8	61.0	15.2	2.11	0.25
50	VBO	315.2	38.0	3.87	0.37	0.038
	Inst	306.7	40.9	4.1	0.392	0.039
	GPUGeom	-	-	-	-	-
	Raycast	117.1	45.5	8.85	1.09	0.118

Basically, all rendering methods scale linearly with the number of particles. There are two exceptions: the values for 10K particles and the values for ray casting. Concerning the first exception, for small data sets the overhead of the individual methods (e.g. even back buffer clearing) becomes the limiting factor as the GPU is not fully utilized. As for ray casting, this approach is, in contrast to all other methods, fragment-processing bound. As the number of particles increases, the particle sizes in image space decrease, reducing the number of fragment operations, which yields better than linear scaling behaviour.

The *VBO-based* rendering achieves almost the same performance for crystallites with 4 to 20 faces. This is probably due to the high CPU load, which is the limit-

ing factor. For 50-faced crystallites, the GPU becomes the limiting factor and the performance drops (compared to 20-faced crystallites) as expected.

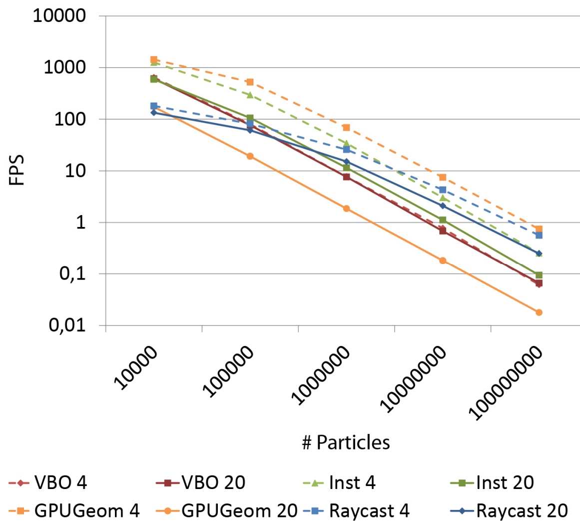

Figure 38: Rendering performance of all methods in fps for tetrahedral-shaped crystallites (dashed lines) and 20-faced crystallites (solid lines); Geometry shader is fastest for 4-faced crystallite (orange line).Most methods decrease linearly with the number of particles, except for when rendering very few particles. Ray casting scales (blue lines) better than linear scaling and thus are beneficial for large data sets.

Hardware-supported instancing trades the high CPU load for the additional overhead of texture upload of the particle data. Surprisingly, despite of this upload, the performance of this approach is favourable in almost all cases, except for complex particles (20 or 50 faces) in small data sets (10K particles). For these cases, the texture upload limits the overall performance. But even in these situations the performance values are comparable to the VBO-based approach.

The *geometry shader* is known to perform poorly when the number of output primitives varies strongly or when the number of output primitives is much higher than the number of input primitives. In the presented approach, the first point is not an issue, as one shader is compiled for each crystallite template and thus the number of output primitives is fixed. However, the second issue can be seen clearly in the performance results. For tetrahedral-shaped crystallites, the geometry-shader-based method clearly outperforms all other methods. For more complex crystallites, however, the performance quickly drops, although it still shows better scaling behaviour than VBO-based rendering (due to the mentioned

CPU limitation). For 20-faced crystallites, the input-output ratio of elements of the geometry shader is disadvantageous to such an extent that the performance values are the worst of all approaches.

The ray casting method is the only approach which is based on fragment processing instead of vertex processing. As described above, it thus shows sub-linear scaling due to the limited screen resolution. For large particles the method is much slower that the other approaches. However, even for the 100K data set the particles become small enough in screen space (still roughly 20×20 pixels) that this method yields performance values comparable to the other methods. Rendering 10^6 or more particles the ray casting reaches frame rates 2-5 times higher than the alternative methods.

As rule of thumb, for data sets up to 1 million particles, instancing yields the best performance results. For larger data sets ray casting is fastest. An optimized hybrid implementation could be obtained when choosing one of these approaches depending on the data set sizes and image-space foot-print of the particles. However, such an implementation would be quite complex, and considering the fact that ray casting is always interactive, it is dubitable if this effort can be justified.

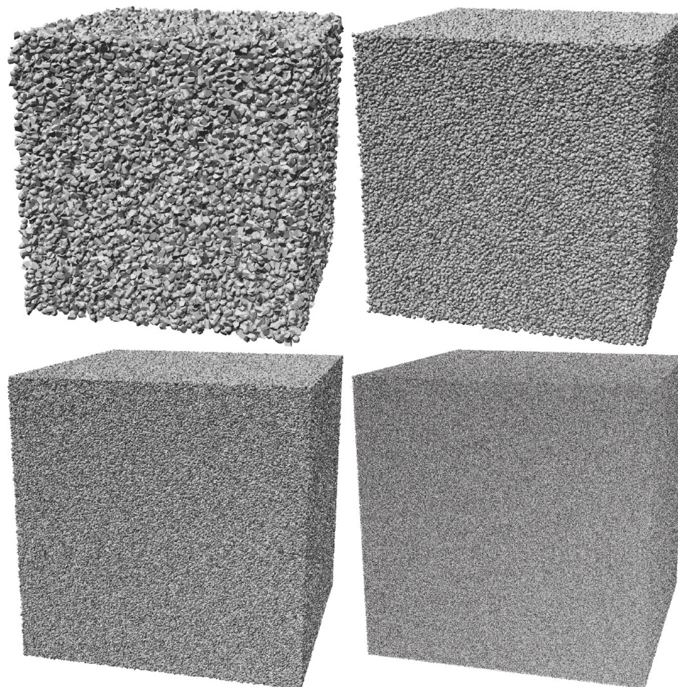

Figure 39: Some data sets used for the performance measurements (cf. Table 12) with (from top left to bottom right) 10^5, 10^6, 10^7, and 10^8 particles. The used crystallite types always have 20 faces.

3 Derived Feature Visualization

Particle-based visualization for particle-based data sets is the obvious first step for a visual analysis process, as it directly shows the content of a data set. The methods presented in chapter 2 allow to interactively visualise data sets with millions of particles on standard workstations. However, as mentioned in the introduction to particle data sets, the individual particles are only one aspect to be analysed. Structures formed by multiple particles and their dynamic behaviour are also in the focus. These structures are usually hard to perceive and track in particle-based visualization, even when using visual enhancements (cf. chapter 2.4). Thus, whenever possible, these structures should be automatically detected and presented in a more continuous and concise way. However, this is not a replacement for the glyph-based direct visualization of the particle data, because such an extraction will remove information from the visualization. Instead, these derived visualizations are supplements. The particle data is always required as reference and ground truth, since only this method guarantees the depiction the original data.

For particle-based data sets from molecular dynamics data the most common structures implicitly formed by the particles are interface surfaces, either between different kinds of particles or between the bulks of particles and surrounding vacuum. Depending on the field of application the data sets originates from, different definitions for such surfaces can be used. In the following sections, surfaces on particle clusters in the context of thermodynamics and proteins from biochemistry are described. As a more specialized example, surface representations of stacking faults and dislocations in solid material are discussed from the field of crystallography in material science.

To study the dynamics of particle data sets, it is often insufficient to simply visualize the trajectory as animation. Visual elements, like path lines, connecting different time frames in still images, provide a much better tool for the qualitative analysis if the number of these lines does not clutter the image. Thus two approaches to apply the concept of path lines to the analysis of the dynamics of large particle data sets are detailed in the following subsections. As an example for visualizing the time-dependent behaviour of derived structures, the dynamics of molecule clusters in the context of thermodynamics is discussed.

3.1 Extraction of spatial Structures

A first step towards creating a derived visualization is to define the features of interest in relation to the original particle data. Interface surfaces between different kinds of particles and between the bulk of particles and surrounding vacuum are often of high interest. The concrete definition of these surfaces is, however, often not that easy. For example, in context of thermodynamics the radius of the spherical glyphs used to depict simple molecules is given by a corresponding influence

radius from the simulation force field (i.e. the Lennard-Jones radius). There the spheres only overlap slightly and naïve surface definitions on that basis would not generate any benefit. Attributes to base such surface definitions on must be identified by the corresponding application domain experts and cannot be generalized to be applicable to all cases. Thus, close collaborations of the application domain scientists and the visualization researchers are required to reach optimal results.

The principal idea is to define an attribute on the particle data or a subset of particles to base the surface definition on. A meaningful attribute is based on simulation attributes in context of the corresponding application. The surface visualization itself can then be generated by different approaches known to the visualization and rendering community, which however can be optimized, based on the application. In this section the focus lies on the definition and extraction of such surfaces from particle data, which can be considered to be static. For time-dependent data, these extractions can basically be performed for every time-frame independently.

3.1.1 Ellipsoid Representation for Particle Clusters

The definition of particle clusters must usually be given by the application domain. Of course, a cluster could be defined by local density of particles, or such, which is a criterion actually used by some applications, but this definition does not have to be meaningful for all application domains. In thermodynamics the application of nucleation, the condensation of vapour to liquid, is a relevant research topic, as this process is found in many physical phenomena, e.g. formation of atmospheric clouds or the effects found in steam turbines. Key properties for the application are the nucleation rate and the critical cluster size. The nucleation rate is the number of clusters reaching the critical cluster size within a given time. The critical cluster size is the number of molecules for which a cluster has the same probability to grow as to shrink. Larger clusters will more likely continue to grow by aggregation of molecules from the surrounding bulk or by coalescing with other clusters, while smaller clusters will more likely shrink by evaporation. Both values depend on a meaningful definition of a molecule cluster, which should be a nucleus, a predecessor for liquid droplets. There exist several definitions for molecule clusters, each with different benefits and disadvantages, defined on geometric properties as well as energetic properties. Some details on such cluster detection criteria can be found in the original publication [GRVE07] and in [HVB+08].

A simple visualization emphasising the molecule clusters is using a single glyph to represent the cluster as a whole. Since these molecule clusters represent droplets the surface tension will force them into nearly spherical shape as soon as the cluster reaches a sufficient size. Using ellipsoids as representing glyphs is therefore an obvious choice. Shaded ellipsoids provide a better perception of the clusters

size and location, even when partially occluded by a high number of surrounding molecules (cf. Figure 40).

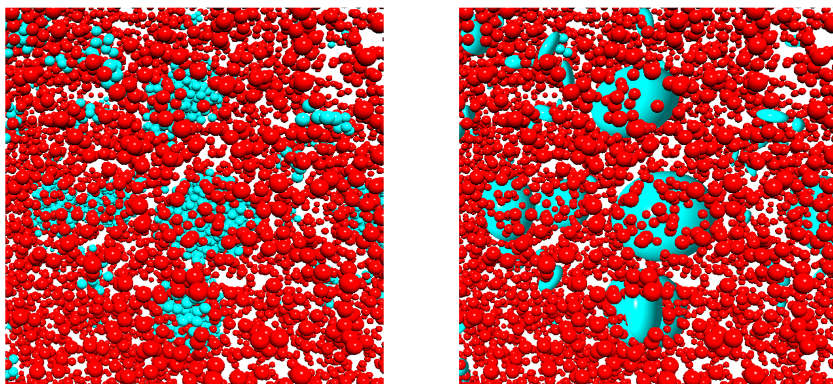

Figure 40: Molecular clusters are normally represented using a simple color coding (left). Using ellipsoids allows easier perceiving the shape of the clusters (right)

The parameters for the ellipsoids are calculated using PCA, similarly to [SGEK97]. The position of the centre of the ellipsoid is the average position of all molecules forming the cluster. The main axes are given by the eigenvectors of the covariance matrix of the positions of these molecules relative to the chosen centre. The radii along these axes are given by the projections of the molecules' relative positions onto these axes. The resulting ellipsoid will miss some border molecules, but with the assumption of nearly spherical shape for the cluster as a whole, this is negligible.

As mentioned earlier, this extraction can be performed for each time frame of a trajectory independently and back-references from clusters to the forming molecules allow for easy tracking of the clusters over time. However, when showing the data as animation and interpolation is needed, interpolating ellipsoids is not trivial, as in addition to ambiguities because of the symmetry of the ellipsoid (axes can be flipped, although the ellipsoid is not rotated), an interpolation can either minimize the scaling or minimize the rotation of the ellipsoid. The first approach will relate the two ellipsoid parameter sets from two time frames based on the radii of the ellipsoid, resulting in the axis with the biggest radius remaining the axis with the biggest radius. The orientation quaternions are then directly interpolated using SLERP [DKL98]. The second approach will correlate pairs of axes to minimize the rotation. Then the radii of the axes are normally interpolated. The differences between these two interpolation methods are only visible if very unstable clusters are used, and then both approaches result in rather extreme animations, either spinning or wobbling.

This representation of molecule clusters works well, if the clusters really are of nearly spherical shape. This is often the case, aside from very special simulations like the one shown in Figure 4 (left). However, even for simulations which behave well in this aspect, small clusters only consisting of a few molecules do not have near-spherical form yet. But exactly these clusters might be of high interest when analysing nucleation. So, although the cluster definition works fine, a better visual representation has to be used.

3.1.2 Particle Clusters as Image-Space Metaballs

One possibility to better depict the true shape of molecule clusters which are not of elliptical or near-spherical form is to rely directly on the particle positions. Metaballs, originally presented by Blinn [Bli82], are defined as isosurface of a density function defined by multiple positions, i.e. the particles. Figure 41 shows an example of nucleation of super-saturated argon, which clearly shows the superior surface shape representation compared to simple ellipsoids.

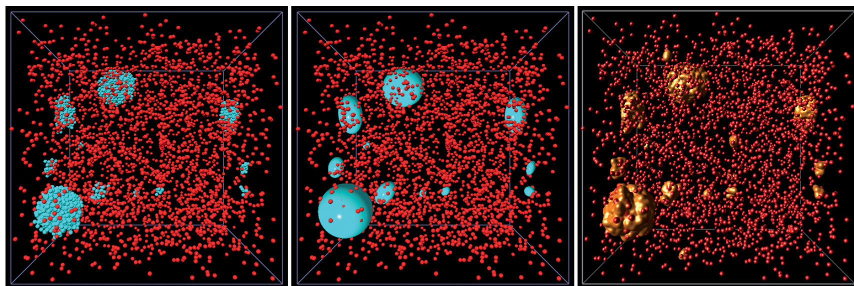

Figure 41: Point-based visualisation of an *argon* nucleation simulation data set. The left image highlights molecular clusters using different colours; the middle image uses ellipsoids and the right one uses metaballs. The metaball visualisation shows closed surfaces of molecule clusters while not exaggerating their size.

Blinn defined the metaball technique for displaying molecular models. The idea is to visualise an isosurface of a simulated electron density field. Blinn originally used an exponential function as field function, which was derived from the density function of a hydrogen atom:

$$D(r) = e^{-ar^2}$$

with r being the distance of the current sampling point to the centre of the current metaball. The sum of the density functions of all atoms then yields the value of the density field.

Blinn originally called his surface *blobs*. The notion of *metaballs* was introduced by Nishimura et al. for the same concept [NHK+85][NN94], who used a piecewise quadratic field function. Several other field functions have been proposed later, like a degree six polynomial by Wyvill et al. [WMW86] or the degree four polynomial by Murakami et al. [MI87][NN94]:

$$D(r) = \left(1 - \left(\frac{r}{r'}\right)^2\right)^2$$

with r' being the support radius of the influence sphere of the current metaball and assuming $D(r) = 0$ for any $r > r'$. In the original publication of the methods presented in this section (cf. [MGE07]) a version of this density function was used which was scaled by 16/9 for a doubled r' (cf. Figure 42). This is motivated by the idea to generate a visually coherent impression compared to ray casting of spheres, i.e. $D(r') = 1$ if only one particle is in range.

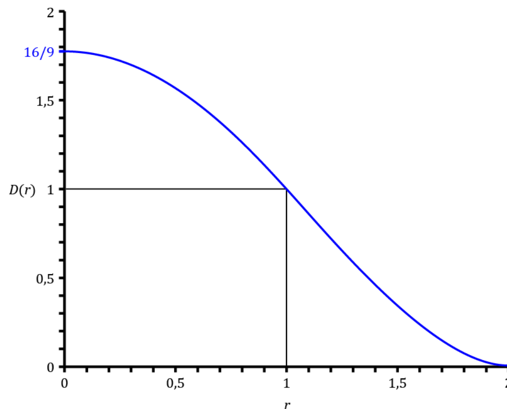

Figure 42: The density function used for the metaballs evaluation presented in this section; An isosurface of value 1 will yield the same impression as ray casting spheres if all particles are sufficiently far away from each other.

A typical approach for visualizing metaballs, which was also chosen by Blinn, is to extract the isosurface, e.g. using Marching Cubes [LC87]. Nvidia showed how to use vertex and geometry shader to evaluate the density field function and generated the corresponding geometry on the GPU [Nvi]. Kooten et al. [vKvdBT07]

employed a point-based method for visualising metaballs on the GPU. They render a large number of particles to approximate the surface. Using repulsion forces between the particles, velocity constraints to force the particles onto the implicit surface of the metaball and spatial hashing allow for optimized evaluation on the GPU. Using insufficient particles, however, can result in poor surface representations and visual artefacts. In their DirectX SDK [MSD], Microsoft showed an image-based approach for achieving a metaball-like effect, by accumulating the density function and surface normal vectors by additive blending in FBOs. As this approach has no depth information, it is limited to a single plane.

To overcome this problem the missing information needs to be made available to the shader creating the surface. A *Vicinity Texture*, storing neighbouring particles for each particle, can be used for this. It allows for rendering the metaballs in a point-based fashion as for compact glyphs, by addressing the main problem when trying to evaluate 3D metaballs in image space: occlusion (cf. Figure 43). Even knowing in advance that, e.g., P1 will not contribute to a metaball, it completely occludes P3. Consequently, when accumulating the density with the influence sphere of P2, without additional vicinity information the contributions of P3 will be missing and the correct metaball (red, dotted lines) will not be shown.

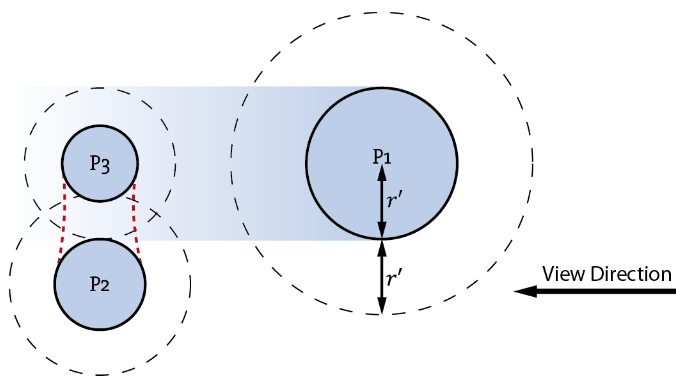

Figure 43: Occluded fragments hamper the image space computation of metaballs: the dashed influence sphere of P3 is completely occluded by P1. Therefore, it is not possible to accumulate the density function correctly in image space. As the contribution of P3 is completely missing, the dotted metaball surface cannot be found and P2 appears as normal sphere.

To make this information available on the GPU, e.g. Kooten et al. [vKvdBT07] use a reversed version of the spatial hash table presented by Teschner et al. [THM+03] to obtain all particles relevant for a specific portion of the screen. Depending on the screen size and grid resolution this hash table can become quite

big. When aiming for optimization of large viewports, it is better to attach the vicinity information to the particle's vertices themselves [MGE07]. This data structure is view-independent and could be pre-calculated or updated only if the particle data itself changes. In a pre-processing step this texture is calculated on the CPU. For each vertex all vertices are stored which are close enough to contribute to the density field. All spheres which do not have neighbours with which they potentially form a metaball are omitted and are ray cast directly as normal spheres.

The rendering of a metaball is then carried out in a single ray casting pass: the vertex shader estimates the image-space foot-print for the influence sphere ($2R$). For each of these fragments an iterative ray casting is performed to find the first occurrence where the iso value is reached. The density field within the influence sphere is sampled by accumulating the density functions of the current particle and all particles specified by the vicinity texture on-the-fly. A reasonable sampling step size needs to be used and if the iso value threshold is missed (sampled value is above) a iterative bisection of the previous sampling step allows to quickly find the correct position. The surface normal vector at the found hit point can be computed as a weighted sum of the normal vectors of the influencing spheres and lighting operations and OpenGL-conform depth values can be calculated analogous to glyph ray casting.

Table 13: Performance values of the *Vicinity-Texture* approach: The processing time specifies the time needed to build up the required data structure texture (view-independent) using a plane-sweep algorithm; Number of potential metaball members in the different data sets: Non-empty groups are the spheres which have at least one neighbour which is close enough to form a metaball with. Spheres which can never form a metaball are counted as empty groups. The size column shows the minimum, average and maximum vicinity group size in the data set.

Data Set	# Spheres	Processing Time	Non-Empty Groups	Empty Groups	Size Min/ Avr/Max	fps
Argon	5000	113 ms	3684	1316	1/39.3/125	0.1
Ethane	25000	2568 ms	24838	162	1/85.2/178	0.2
Sim-3	100	<1 ms	99	1	2/8.0/17	4.0
1A2J	1454	12 ms	1454	0	2/6.9/13	0.14

Table 13 shows the performance of the vicinity-texture approach. The tests were run on an Intel Core2 Duo 6600 processor with 2.40 GHz, 2 GB memory, and an Nvidia GeForce 8800 GTX graphics card with 768 MB graphics memory. The viewport size was 512×512 for all tests. Four data sets were tested with this approach: The *Argon* data set (Figure 41) is a nucleation simulation of argon with 5000 molecules. The *Ethane* data set (Figure 44, right) consists of 25000 molecules and shows a nucleation process in a supersaturated configuration. The *Sim-3* data set (Figure 46 and Figure 47) is a generated test data set used while developing our software. It

was created using random placement, only controlling that the overlapping of spheres is limited. The data set *1A2J* (Figure 44, left), is a disulphide bond formation protein imported from the protein data base [BWF+00] as example of the applicability of this method to other kinds of data sets.

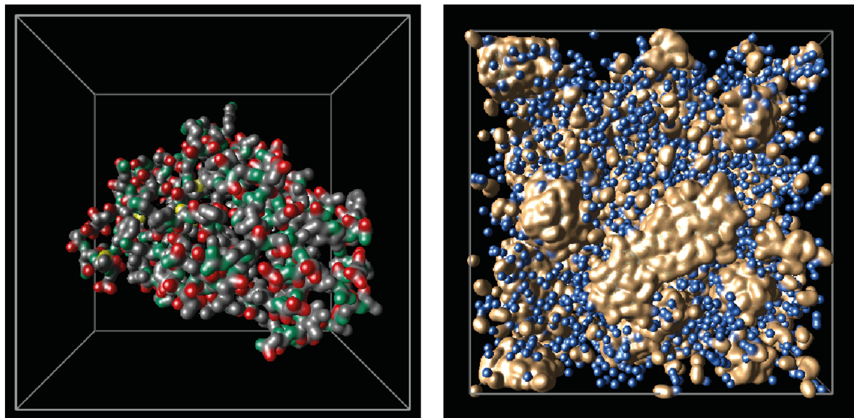

Figure 44: Additional data sets tested with the metaball technique: left: disulfide bond formation protein *1A2J* from the protein data base [BWF+00]; right: a nucleation simulation of supersaturated *ethane*.

The vicinity texture method performs poorly, even for these rather small data sets. One main problem is the tremendous amount of texture fetches required during the on-the-fly evaluation of the density function. While this has improved with newer-generation GPUs, another problem is shown by the number of spheres potentially contributing to a metaball (having non-empty groups). E.g. data set Sim-3 has nearly only metaballs and the number of texture accesses is very high, which is always a problem, regardless of the GPU generation. The protein 1A2J is the extreme case which only forms one single metaball.

However, one interesting fact about the vicinity-texture approach is that it scales quite well with the image resolution. Preliminary tests with a 1600×1200 viewport yield almost the same performance values as with the usually used 512×512 viewport. This is an indicator that the texture fetching and caching capabilities of the graphics cards are still not fully utilized. Thus the main reasons prohibiting good frame rate is the algorithmic complexity of the two nested loops performing the iterative sampling, especially because the number of neighbouring spheres is read from the vicinity texture and the number of loop iterations varies strongly among the particles. The resulting divergence of parallel computation threads within the GPU reduce the overall processing power, which is an effect recently studied by Frey et al. [FRE12].

Trying to remedy this problem, a second, orthogonal approach can be chosen: instead of generating few but expensive fragments, many but cheap fragments might be better able to utilize the graphics hardware. Inspired by depth peeling [Eve01], the final image is evaluated in image-space by moving an imaginary plane aligned to the viewport through the data set in several rendering passes. The method is thus called *Walking Depth Plane* and is sketched in Figure 45. Two FBOs are used to store the needed data. The first FBO (called λ-buffer) stores the depths of the fragments, which approximate the targeted isosurface in image space. In addition to this distance from the position of the camera in world-space coordinates, a maximum distance values, based on the data set's bounding box, is stored and later used as termination criterion. The second FBO (density buffer) is used to evaluate the density field at the depths provided by the λ-buffer. Kanamori et al. [KSN08] presented a method with a similar fundamental idea. They used Bézier Clipping to find the ray-isosurface intersection. Linsen et al. [LvLRR08] extended this approach to multivariate data.

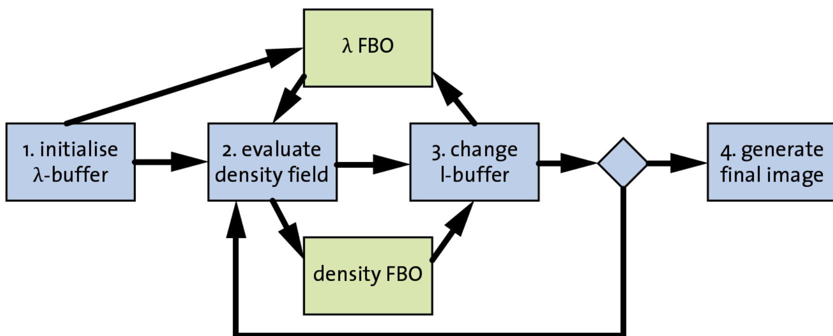

Figure 45: The different rendering passes of the *Walking Depth Plane* approach, together with the two frame buffer objects involved.

First the λ-buffer is initialized using a modified ray casting shader for the influence spheres of all particles. The front hit point of the front-most sphere is used as starting value, while the back hit point of the back-most sphere is used as termination value. The iterative rendering will terminate if the current λ value is larger or equal to the termination value. Pixels which are never written by any fragments are initialized by *glClearColor* values to be instantly terminated.

For all other fragments, until the termination criterion is met, the steps 2 and 3 are repeated: In step 2 the density field is evaluated. The particle data is drawn and for each fragment the density contribution of the corresponding metaball at the position defined by the depth specified by the λ-buffer is added to the

density buffer utilizing additive blending. Using the separate channels of the density buffer, the density can be evaluated at multiple depths at once, similar to the method presented in [NM05] for performing RBF-based volume rendering. In step 3 the values of the λ-buffer are updated, again using additive blending, to actually move the plane towards the targeted isosurface. The direction of the step is given by the value in the density buffer: i.e. a value below the threshold will result in a forward step. The radius of the smallest sphere can be used as maximum step size. Similar to the recursive bi-section used in the vicinity-texture approach, the step size must be adjusted when the isovalue is approached. However, to avoid ping-pong rendering between two λ-buffers or the need for read-write access to a single buffer, a simple oracle can be used to control the step size. As approximation a quadratic attenuation for steps into the viewing direction and linear attenuation for steps in opposite direction are used for the step size Δ:

$$\Delta = \begin{cases} \Delta_{max}\left(1 - \widetilde{D}(x)^2\right) & \text{for } \widetilde{D}(x) < 1 - \epsilon \\ 0.5\Delta_{max}\left(\widetilde{D}(x) - 1\right) & \text{for } \widetilde{D}(x) > 1 + \epsilon \\ 0 & \text{otherwise} \end{cases}$$

with Δ_{max}being the maximum step size, ϵ an approximation tolerance, and $\widetilde{D}(x)$ the summed density over all metaballs at position x. Figure 46 shows the changes in the λ-buffer as the algorithm is executed.

The loop of the algorithm is left as soon as one termination criterion is met. As the maximum step size is known, the required number of iterations can be computed using this value and the data set's bounding box. Since, on one hand, this value is usually highly overestimated and, on the other hand, the oracle can decrease the step size, this value is adjusted (clamped to heuristically chosen constants) before it is used as maximum number of loop iterations. As second control mechanism to ensure responsiveness, a maximum execution time for the algorithm is defined, which guarantees a minimum frame rate at cost of worse approximation of the isosurface.

Finally, each fragment can be tested if it is terminated, which is shown in Figure 47. If the value is sufficiently close to the targeted isovalue the pixel is marked as finished by setting the value of the depth buffer of both FBOs to zero, resulting in the depth-test removing any fragments in both calculation steps. As both FBOs share a common depth attachment, this is accomplished by a single operation. As soon as a given percentage of all pixels is marked as finished, the iteration loop is terminated.

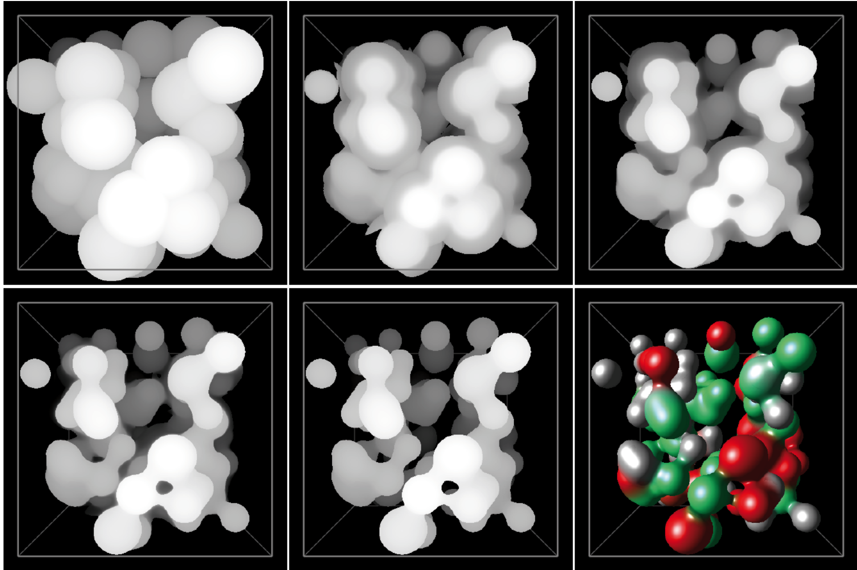

Figure 46: Metaball isosurface approximation in *Sim-3* data set after different number of iterations. Lower right image shows final output image of the *Walking Depth Plane* approach. The other images show the content of the λ-buffer after different numbers of iterations (from upper left to lower right): directly after initialisation, after 5, 10, 15, and 31 (final result) iterations.

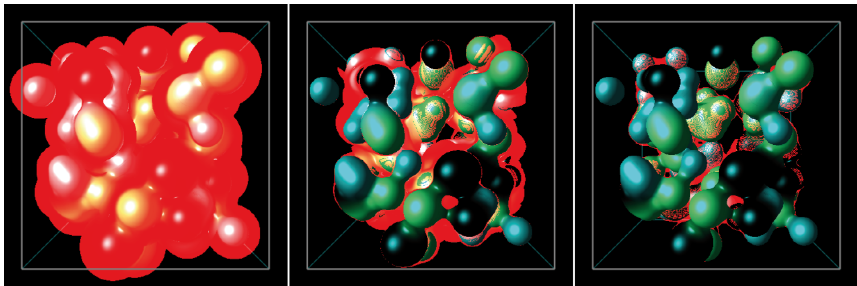

Figure 47: The metaball of the data set *Sim-3* after 5, 10, and 30 iterations (from left to right). The red colour channel encodes the finished pixels black (showing the isosurface in green/cyan) and unfinished pixels red.

Table 14: Performance values of the *Walking Depth Plane* method. The maximum number of iterations is changed for comparability or for artefact-free visualization.

Data Set	Max. # of Iterations	fps
Argon	100	13
Ethane	100	5
Sim-3	100	15
Sim-3	31	61
1A2J	100	21
1A2J	250	11

Table 14 shows performance values for the *Walking Depth Plane* approach. The tests were conducted on the same hardware with the same data sets as for the measurements in Table 13. Since the approach does not only depend on the complexity of the data set, but also on the maximum number of iterations, we provide rendering performance values for all data sets using exactly 100 iterations. Increasing or decreasing the maximum number of iterations intuitively decreases or increases the frame rates.

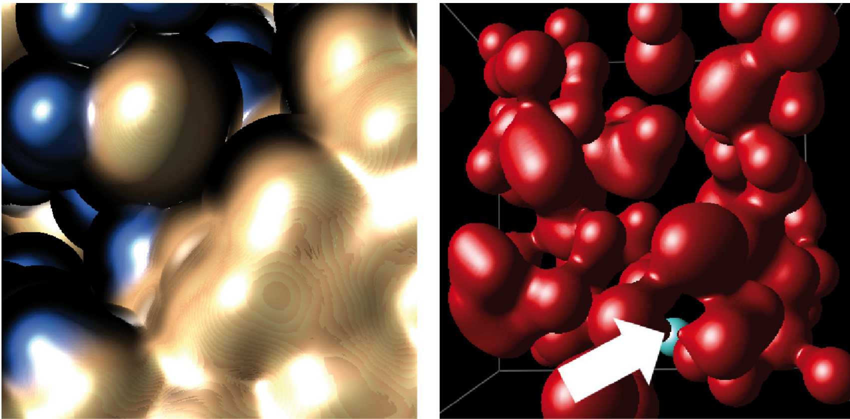

Figure 48: Visual artefacts: left: under-sampling artefacts occur when rendering the ethane data sets with the *Moving Depth Plane* and using too few steps; right: Missed surface parts (arrow) due to under-sampling by the *Vicinity Texture* approach.

If the number of iterations is limited and the algorithm cannot reach its isosurface (1A2J with only 100 iterations) visual artefacts appear (cf. Figure 48, left): black border regions as well as step-band artefacts on the isosurface. The latter are under-sampling artefacts as the λ values approach the targeted isosurface from both sides but were not able to reach the value yet. Similar under-sampling arte-

facts are also visible for the *vicinity texture* approach (cf. Figure 48, right) missing a thin connection formed by metaballs.

3.1.3 Isosurface in Particle Density Volume

An optimization of the *Walking Depth Plane* can be achieved by performing volume ray casting between planes of evaluated density values. This reduces the accuracy issues on the isosurface approximation because of the faster convergence to the targeted isovalue. This optimization was originally presented in [FGE10]. The core idea is to (partially) construct the density volume and then perform ray casting to generate the isosurface. The system described by Cha et al. [CSI09] also follows this approach. Schmidt-Ehrenberg et al. [SEBH02] used a similar method to visualize the flexibility volume in the context of biochemistry. Isosurfaces in volume rendering are usually obtained by correspondingly set transfer functions. Different techniques of direct isosurface rendering were developed over several years. Starting with blended texture stacks (e.g. [WE98]) volume rendering evolved using 3D textures and multi-pass volume ray casting (e.g. [KW03], also cf. [DCH88]) finally to single-pass ray casting (e.g. [SSKE05]). Volume splatting, which is related to the density volume reconstruction for metaball surfaces, was proposed by Westover [Wes90] and optimized to modern GPUs, e.g. by Neophytou et al. [NM05]. The surface rendering optimization presented in this section is related to these volume rendering methods. The volume is reconstructed using splatting directly in image-space at view-port resolution.

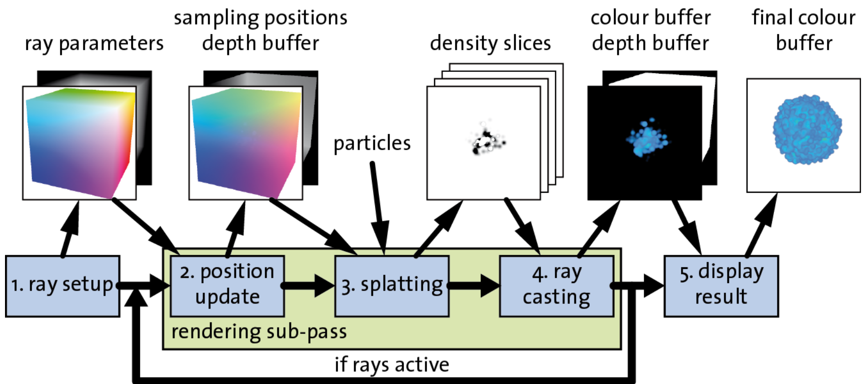

Figure 49: Flow chart of the sliced ray casting method; After initial ray setup, multiple rendering sub-passes of sampling position update, density reconstruction through splatting and volume ray-casting are performed to generate the final image.

The viewing volume is subdivided into concentric spherical shells centred at the camera position. The shells are equidistant at the volume sampling distance, starting and ending at the extent of the data set's bounding box, and are used to reconstruct the density volume, similar to the λ-buffer in the *Walking Depth Plane* approach. However, as the reconstruction is performed on concentric spherical shells, the λ values are given implicitly by the pass' number and don't need to be stored explicitly. The whole algorithm is depicted in Figure 49.

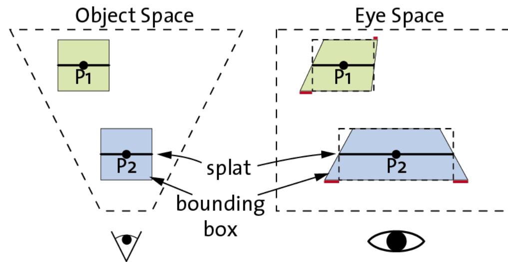

Figure 50: Perspective correction is necessary when projecting splats from object space into image space. Otherwise the splats are not fully represented in image space.

In each rendering pass of the loop, the density volume is reconstructed by splatting perspective-correct density kernels for the particles into the density FBO. These kernels are similar to the rotationally symmetric functions used by other metaball approaches. Since we evaluate the density volume in image-space, the splatting of these kernel functions must be perspectively correct, as can be seen in Figure 50, otherwise the kernels might get truncated (red areas). The splatting is performed by classical point-based rendering, same as glyph ray casting, and the density accumulation is implemented by additive blending. Utilizing multiple render targets allows several depth values to be reconstructed at once into a stack of slices and allows reducing the number of iterations and thus the number of particle draw calls. Using slices with uniform depths, the particles not contributing can be efficiently culled. The culling can be further optimized by sorting the particles along the viewing direction, e.g. in CUDA using the available Radix Sort. Then only the particles need to be drawn which will contribute to the current depth FBOs. This is mainly possible, in contrast to the walking depth-plane approach, because the slices are equidistant, i.e. move with the same speed through the data set.

Within a stack of density slices single-pass volume ray casting is used to render the isosurface. For a continuous isosurface, the density between the slices is interpolated and the last slice of the previous stack is re-used as first slice of the current stack. The result of the different rendering passes is composited to obtain

the final image. For opaque surfaces, the depth buffer for the sampling positions is updated to terminate the iterations for fragments hitting that surface.

Since the sub-passes of our approach are CPU-controlled, but the states of the individual rays exist on the GPU, a feedback mechanism is required. HWOQs are used to test the plane of viewing rays against the depth buffer. The number of active viewing rays is then evaluated on the CPU. To hide the latencies, we exploit the frame-to-frame coherence and postpone the evaluation of the query result and the adjustment of the maximum number of sub-passes for the next frame.

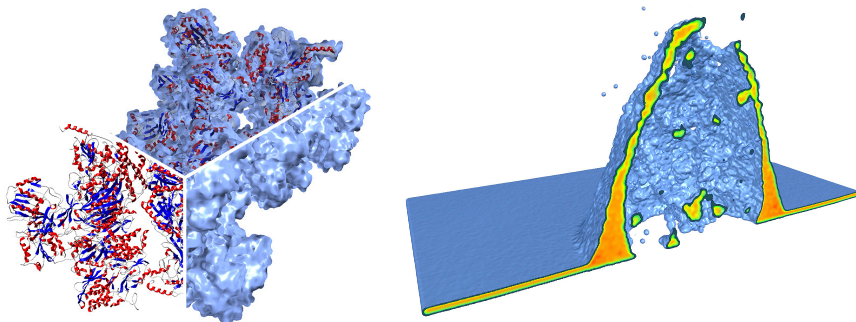

Figure 51: Two example visualizations created with the density volume isosurface approach; Left: Visualizations of the oxidoreductase protein with ~75,000 atoms (~10,000 amino acids). The lower left sub-image shows a cartoon representation of the secondary structure. The lower right sub-image shows the described surface volume rendering while the top sub-image shows a combination of both. Right: Laser ablation simulation showing a bulge on a metal block. The cutaway of the bulge reveals the particle density distribution inside.

Figure 51 show two example data sets visualized with this method. The left image shows visualizations of a protein which closely resemble the Molecular Skin Surface (MSS; cf. [CLM08]) used in protein visualization. The right image shows the result of a MD simulation of laser ablation with roughly 250,000 particles. On an Intel Core2 Duo 2.4 GHz with 2 GB RAM and a Nvidia GeForce 280 GTX with 1GB RAM the laser ablation data set, which has roughly 10 times more particles than the previously tested *Ethane* data set, can be rendered with 1.6 fps for a viewport of size 1500×900. However, because of the nature of volume ray casting in combination with the density volume being reconstructed at image-space resolution, no visual artefacts as shown in Figure 48 or Figure 52 (left; volume reconstructed in object space prior to standard isosurface ray casting) occur. Thus, the image quality is enhanced and the rendering method is applicable to small and medium-sized data sets (e.g. common in the field of biochemistry).

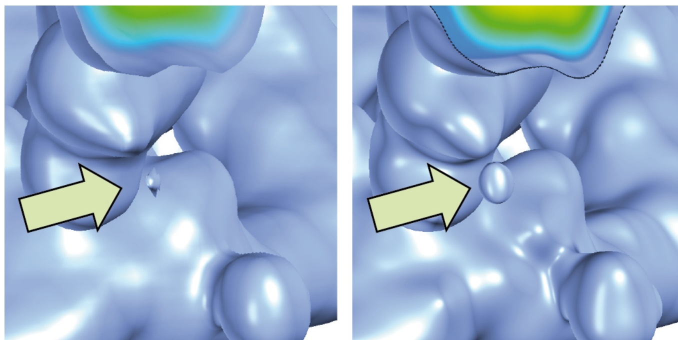

Figure 52: Close-ups of a small region of the laser ablation data set shown in Figure 51: volume construction in (left) object space or (right) image space. Ridges, grooves, highlights, and small features are more distinct with image-space reconstruction.

3.1.4 Molecular Surface

Other research fields in which MD simulations are commonly used already have established visual metaphors for concise, derived representations. One related example are surface representations for macro-molecules like proteins in the context of biochemistry, often used when studying molecule-molecule or molecule-solvent interactions, e.g. in drug design. There are several possible definitions of a molecular surface: the Van-der-Waals (VdW) surface probably is the simplest definition in which each atom is shown by a sphere with the corresponding VdW radius. The major drawback is that the VdW surface shows the general shape of the molecule, but does not model regions accessible to solvent molecules.

Richards thus defined the Solvent Accessible Surface (SAS) in 1977 [Ric77], which includes a spherical probe representing potential interaction partners. As this probe rolls over the VdW surface, its centre traces out the SAS. Minor gaps and crevices not accessible by the solvent molecules are closed, but the apparent inflation of the SAS makes it difficult to perceive the positions reachable for the solvent in relation to the internal structure of the molecule. Richards also introduced a Smooth Molecular Surface, which is traced by the surface of the probe sphere. Greer and Bush [GB78] defined this surface to be the union of all possible probes that do not intersect with the atoms of the molecule, from which the name for this surface definition was derived: Solvent Excluded Surface (SES). The SES combines the advantages of the VdW surface and the SAS. Figure 53 shows a schematic of all three surface definitions.

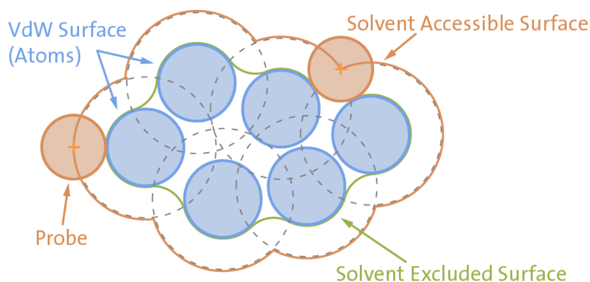

Figure 53: 2D-schematic of the molecular surface definitions. The rolling probe traces out the SAS and the SES.

While rendering the VdW surface and the SAS is simple and straightforward via glyph-based visualization, showing the SES is not trivial. As the probe roles over the VdW surface three graphical primitives are needed, which were also defined by Richards:

- *Spherical patches* occur where the probe is rolling over the surface of a single atom and has no contact with any other sphere.
- *Toroidal patches* are formed when the probe has contact with two atoms and rotates around the axis connecting the atom centres.
- *Spherical triangles* are generated by the probe's surface when the probe is simultaneously in contact with three atoms.

Connolly [Con83] presented the first algorithm for computing the SES analytically which was later improved by Perrot et al. [PCG+92]. Calculating the three types of graphical primitives based solely on the atom positions is still computationally expensive. Varshney et al. [VBW94] proposed using an approximate Voronoi diagram allowing for a parallelized construction scheme. Sanner et al. [SOS96] presented an algorithm based on a reduced surface definition, which was quickly extended to partial updates for dynamic data [SO97]. Another evaluation of the SES was presented by Edelsbrunner and Mücke [EM94] based on α-shapes, which was recently extended to β-shapes by Ryu et al. [RPK07]. Totrov and Abagyan [TA95] proposed the Contour-Buildup algorithm, which was parallelized by Lindow et al. [LBPH10] for multi-core CPUs. Classical rendering approaches were based on triangulation of the surface, resulting in the well-known trade-off between computation effort and display quality. Efficient calculations were presented e.g. in [Con85], [ZXB07], and [RCK09]. Halm et al. [HOF05] support texture-based, glyph-based, as well as polygon-based rendering including level-of-detail techniques for whole molecule surfaces. A first implementation using only GPU ray casting based on the reduced surface was presented by Krone et al. [KBE09]. Noteworthy is the ray casting of the toroidal patch, which is described by a polynomial of degree four. Improved rendering techniques for higher order polynomial surfaces have been

presented, e.g. in [LB06], [SN07], and [dTLP07]. Toledo and Levy [TL08] use the iterative Newton-Raphson algorithm to solve the viewing-ray-surface intersection for higher order surfaces. Singh and Narayanan [SN09] and Krone et al. [KBE09] apply the analytic stabilized Ferrari-Lagrange method presented by Herbison-Evans [HE95].

The Contour-Buildup algorithm can be parallelized to efficiently run on graphics hardware utilizing the capabilities of GPGPU APIs like CUDA. To fully utilize modern GPUs, an algorithm must be divided into small independent working tasks. Lindow et al. [LBPH10] parallelised the algorithm over all atoms calculating intersections with the neighbouring atoms. This section presents a parallelization over all neighbours of all atoms by slightly changing the algorithm, which is more fitting for highly parallel GPUs. This approach is fast enough to recalculate the whole surface for large proteins interactively. The required graphical primitives are calculated using CUDA, and the final image is rendered by performing point-based ray casting of polynomial surface patches entirely on the GPU. The concept was first published in [KGE11].

The general idea of the Contour-Buildup is to compute the intersection of pairs of spheres of the SAS and derive the SES from them (cf. Figure 54). If more than two spheres intersect each other, the small circles defined by pairwise intersection of spheres may also intersect each other, resulting in circle arcs. The contour is then defined as the union of all of these small circle arcs which lie outside all other spheres. The graphical elements of the SES are derived from this contour: At the two endpoints of a small circle arc the rolling probe is in a fixed position, defining a spherical triangle. The arc itself describes the path of the probe while it traces a toroidal patch. The remaining spherical patches are given by the VdW surfaces of all atoms contributing to the contour.

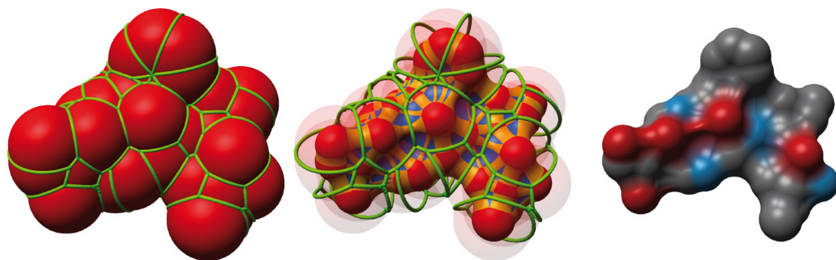

Figure 54: Illustration of the Contour-Buildup algorithm to compute the SES for a small protein (3NVF). The input is a set of intersecting atoms whose radii are expanded from VdW surface by the radius of a probe sphere to form the SAS (left). The algorithm extracts the contours from which the surface patches of the SES can be derived (middle). The right image shows the final rendering of the SES.

In detail, the Contour-Buildup algorithm tests the SAS-sphere of each atom for intersections with all neighbouring atoms resulting in the set of all possible small circles. Each newly created small circle is subsequently intersected with all previously computed small circles until only the visible circle arcs remain. If two small circles do not intersect although residing on the same sphere, one circle can be not visible, and is thus removed, or the atom can be completely covered by its neighbours and is thus removed completely. If the two small circles intersect, an existing arc can either be cut off, shortened, or split into two. The equations to evaluate which of these cases apply are given in [TA95].

The parallel Contour-Buildup algorithm for GPUs does not calculate the whole contour for one atom per thread, but only the remaining arcs for each small circle for each pair of atoms. The number of independent, concurrent calculation tasks is thus increased to better suit modern GPUs. Figure 55 shows the processing steps of our method. The atom positions and radii needed are stored in an VBO which can be accessed from CUDA kernels as well as used for GPU-based glyph ray casting. As first calculation step, the neighbourhood for each atom is computed using the spatial hashing approach given in the *Particles Demo* from the Nvidia GPU Computing SDK [Gre08]. This implementation is a fast, grid-based spatial subdivision which runs in parallel using CUDA (first three kernels, cf. Figure 55). Using the spatial subdivision, all neighbouring atoms of each atom are extracted and stored in a lookup table in parallel for all atoms. The small circles are also computed and stored to a lookup table at this point. A further kernel removes all small circles which are completely removed by successive cutting with other small circles and marks atoms to be excluded from further processing, because they are completely hidden inside the surface. This kernel is executed in parallel for all neighbours of all atoms.

Afterwards, the main CUDA kernel computes the contour arcs (right Kernel, middle row, in Figure 55). Each small circle of each atom is intersected with all other small circles of the corresponding atom and the remaining arcs, if any, are computed. Only the end points of the arcs and the small circles which contribute to the arcs are stored, as only this information is required for the spherical triangles and toroidal patches. A small circle is only marked visible, creating a toroidal patch, if at least one of its arcs remains in the result set. Note that each arc would be computed for both atoms that share the corresponding small circle. However, these redundant computations can safely be removed. As the atoms have an implicit order we can ignore small circles between an atom and its neighbouring atoms with smaller ordering indices. Every endpoint is shared by three arcs, which would also lead to a similar redundancy, which can be removed with the same approach.

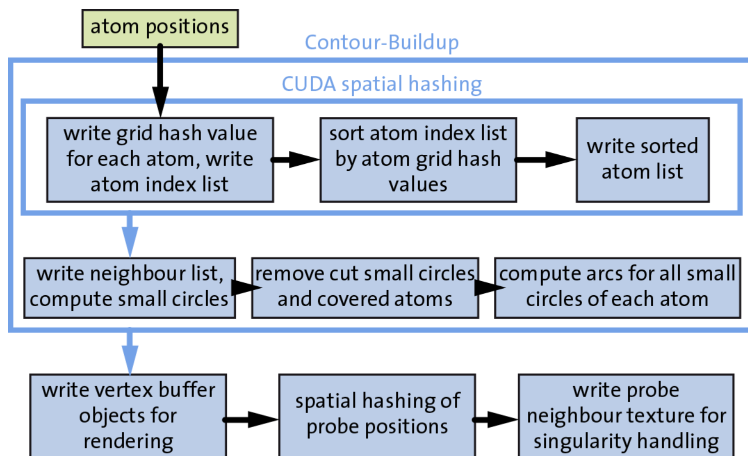

Figure 55: The processing work flow of the parallel GPU Contour-Building algorithm. Each blue box represents a CUDA compute kernel. Note that there is only one single data upload (green box, representing a VBO) for computation as well as rendering.

From these results, the graphical elements required to render the SES are derived and the required parameters are stored in a set of VBOs. The number of toroidal patches and spherical triangles to be rendered is determined using the parallel prefix sum provided by the CUDA Data Parallel Primitives library [cud11], which allows compacting the probe and torus positions when writing to the corresponding VBOs. This is carried out using a further CUDA kernel which is, again, executed in parallel for all neighbours of all atoms. The last step is a singularity handling for the spherical triangles intersecting each other, based on a neighbourhood search using spatial hashing (cf. [KBE09]). Finally, point-based ray casting is used with GLSL shaders to create the final image.

Table 15: Performance measurements for the CUDA implementation of the parallel Contour-Buildup algorithm; *CB* denotes the execution time of the Contour-Buildup algorithm (including data transfer). *VBO+SH* denotes the time for writing VBOs for rendering and the handling of singularities. The column labelled *fps* states the overall performance in frames per second (i.e. data transfer, Contour-Buildup computation and rendering of the SES).

Data set	# Atoms	CB (ms)	VBO + SH (ms)	fps
1OGZ	~1000	9.5	2.1	71
1VIS	~2500	14.1	2.6	50
1AF6	~10000	36.1	7.3	20

This implementation was tested with several data sets freely available from the Protein Data Bank [BWF+00], which are shown in Figure 56. The results are shown in Table 15. The test machine was an Intel Core i7-920 2.66 GHz with 6 GB RAM and an Nvidia GeForce GTX580 (1.5 GB RAM). The viewport resolution was set to 1024×768 pixels. All computations are performed for every frame anew. Therefore, the performance values, even though measured with static data sets, are valid for time-dependent data sets of similar size as well. The computation times for the Contour-Buildup algorithm (column *CB*) include the time required for the data transfer. The data transfer between CUDA and GLSL is given separately (column *VBO+SH*). The results show that the presented algorithm reaches interactive frame rates for medium-sized proteins with tens of thousands of atoms. The work of Lindow et al. [LBPH10], using OpenMP loop parallelization, managed to approximately gain a sixfold speedup on a CPU with eight physical cores, and their implementation scales linearly with the number of atoms. In contrast, the CUDA implementation presented here exhibits better than linear scaling for large data sets. Most probably the presented modifications of the algorithm are better suitable for parallel hardware due to the more fine-grain computation tasks. For the maltoporin data set (1AF6) Lindow et al. measured an update time of 73 ms (2 Intel Xeon E5540 2.53 GHz) while our implementation takes only 43 ms. For very small data sets like the isomerase data set (1OGZ) the presented implementation reaches only slightly faster update times (16.7 ms versus 18 ms), which can be explained by the overhead introduced by CUDA.

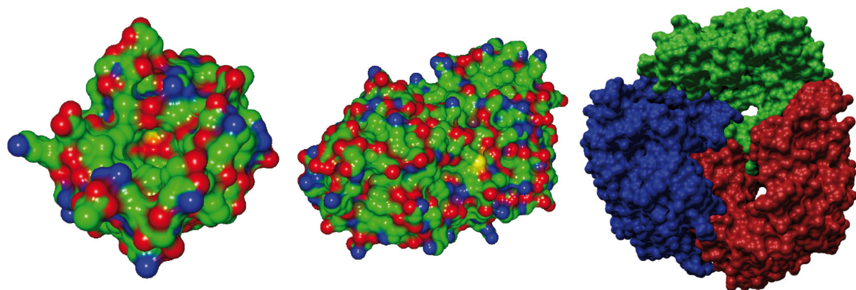

Figure 56: The data sets presented in Table 15; from left to right: 1OGZ, 1VIS, and 1AF6

3.1.5 Linear and Planar Crystal Defects in Atom Lattices

Interface surfaces, as shown in the previous sections, are a common example for spatial structures to be extracted from MD data. However, depending on the application, other structures can also be of interest, like the molecule clusters in thermodynamics (cf. Chapter 3.1.1). Those concepts are related, as the representation of

molecule clusters can also be understood as interface surface between groups or particles or areas within the data set with different attributes (e.g. change in local density). A further example of structures of particles with different attributes is the visualization of crystal defects in the atom lattice of crystalline materials, e.g. solid metals. Such defects occur, e.g., in compound materials or if external forces are applied, and are known to be directly related to macroscopic material properties such as resistance and strength. A common way to describe such atom lattice defects is given by the Burgers vector [BC06], which represents the magnitude and direction of the lattice distortion of a given defect. Schall et al. [SCWS04], [SCWS06] discussed ways to track defects in colloidal crystals, and techniques to visualize the distribution of Burgers vectors (represented by the Nye tensors). Hartley et al. [HM05] also discussed a similar approach that measures Nye tensor distributions.

Defects are basically changes in the regularity, symmetry, or ordering of a 12-atom neighbourhood [Set06]. Such defects align in linear structures (called *dislocations*) and planar structures (called *stacking faults*). Many works on analysing these defects are based on meso-scale representation, e.g. dislocation dynamics simulations. Direct visualizations of such simulation results are presented e.g. by the work of Lipowsky et al. [LBM+05]. Bulatov et al. [BCF+04], [BAK+98] suggests tracking the topological structure of dislocations and special features like junctions and multi-junctions (connections of three or more dislocations), as these are known to have a strong impact on the material properties [BHT+06].

In this section a method is presented to extract dislocations and stacking faults from atomistic MD data sets. Derived visualization of these features allows for much cleaner representation of the structure and topology formed. This method was first presented at [GDCE09]. A similar extraction and visualization was later presented in [SA10].

The method to extract stacking faults and dislocations consists of four steps, depicted in Figure 57. First the neighbourhood graph is constructed, connecting each atom to all atoms within geometric proximity (cf. [Sup83], [AM92]). Then the graph is simplified by iterative contraction, controlled through a simple mass-spring approach, as well as edge collapsing, until the graph is transformed into the topological structure formed by the dislocation. Based on this structure information, the original atom data set is segmented into individual dislocations and stacking faults. Finally, tubes and planes are fitted to the data, to visualise the structure more clearly than the usual glyph representation does.

The main step, the graph simplification, is related to skeleton extraction, which is a vast research field of its own. An introductory survey can be found in [CMS07]. The graph representation as well as its contraction, at least the basic idea, is related to distance-field-based thinning methods. Other skeleton extraction methods are based on geometric methods and proximity structures like Voronoi Diagrams, Reeb Graphs or Contour Trees (cf. [GNP+06], [EHM+08], [CSA03]). The contraction-controlling mass-spring system corresponds to smoothing operations often employed for a similar task (cf. [ATC+08]). General field methods, such as

potential field functions are used in similar applications. In the work of [NEA+04] crystal dislocation data is given as a potential field function and a Morse-Smale complex is used to evaluate the structure of the dislocations.

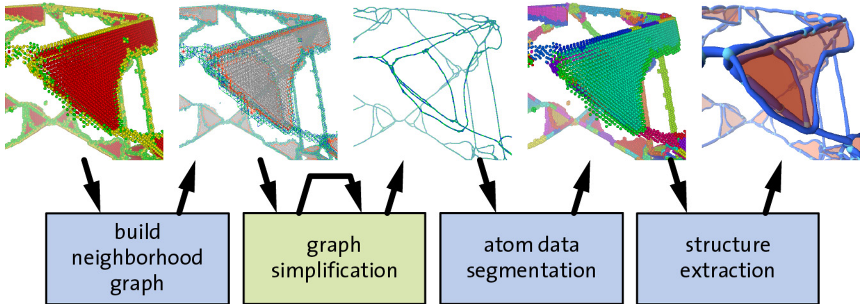

Figure 57: Overview of the method to extract dislocations and stacking faults. From left to right: first the atoms are classified based on their 12-neighbourhood. The neighbourhood graph is contracted and simplified to get the topological structure of the dislocation network. This graph is then used to segment the atom data set. Finally tubes and planes are fitted into the segmented atom positions.

In this section, the dislocations are extracted directly from atomistic MD data. The data set used in most figures in this section (cf. Figure 57 and Figure 63) is a block of compound material of Ni and Ni_3Al, underlying a stretching force. The data set originally contains 1.2 million atoms, but can be filtered to only contain between 13 000 and 87 000 atoms. The classification of the 12-atom neighbourhood according to [HA87] is used for this filtering. Nickel forms an FCC lattice in the undistorted case. These atoms are not of any interest and are therefore removed from the upcoming computations and visualizations. The neighbourhoods of the remaining atoms may either form a BCC lattice, a hexagonal-close packed (HCP) lattice, or may be aligned irregularly. To change a lattice from FCC to HCP one plane needs to be shifted, which is characteristic for stacking faults. Thus, the accordingly classified atoms need only to be considered when extracting stacking faults and can be ignored for the computation of dislocations. Furthermore, as it is known that stacking faults are always surrounded by dislocations [HL92], extracting the planar stacking faults is easy as soon as all dislocations have been found. All atoms either classified BCC or having an irregular neighbourhood structure thus form local defects or dislocations. Additionally, dislocations cannot have open ends. They will always form loops or networks.

To extract the topological structure, first a neighbourhood graph is constructed based on a simple distance criterion using domain knowledge about the uniform atom distances within undistorted atom lattices. Being N the set of all

nodes in the graph, where initially one node $n_i \in N$ is created for one atom a_i each. The position of each node $\boldsymbol{p}(n_i)$ is set to the position of the original atom. The edges between the atoms are stored in three lists depending on the atom classification of the corresponding atoms. E_d holds all edges of atoms which will form *dislocations*, i.e. atoms have BCC or irregular neighbourhood structure. E_s stores edges between atoms with HCP neighbourhoods, which will form *stacking faults*. All remaining edges, which form the *borders* between stacking faults and dislocations, are collected in E_b. The different sets of edges are handled differently in the stages of the algorithm, e.g. to ensure that small stacking fault areas don't get removed by the graph contraction.

This contraction is based on a simple mass-spring system. For each node n_i a speed vector is stored. Nodes connected to edges from E_d are allowed to move. Edges from E_d and E_b are handled as springs. Edges from E_d are parameterised to collapse to zero length, while edges from E_b are parameterised to keep their initial length, to conserve stacking fault areas. Figure 58 shows a result of the graph contraction.

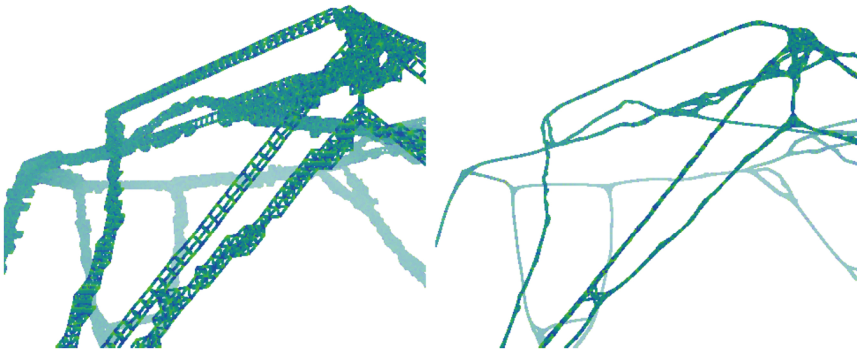

Figure 58: Graph contraction; left initial neighbourhood graph, right after contraction. Only Edges from E_d are shown.

In addition to moving the nodes the graph itself is simplified by collapsing small edges, based on a threshold derived from the neighbourhood distance. Collapsing an edge combines the two nodes n_i and n_j which were originally connected. The new node n_k replaces both, meaning, that n_k references all atoms of n_i and n_j and that all edges originally connected to either one of these nodes are now connected to n_k. $\boldsymbol{p}(n_k) = 0.5\left(\boldsymbol{p}(n_i) + \boldsymbol{p}(n_j)\right)$ and additional information like the speed vector is handled correspondingly.

Additionally, to identify junctions of dislocations, all nodes are classified to be either dislocations or junction nodes. The classification is, at first, based on the directions of the edges connected to the node. The relative positions of all connected

nodes is averaged to gain a support position for n_i. Then the dot products of all pairs of normalized vectors from this support position to the positions of the connected nodes are evaluated. If all results of the dot product are within a small threshold equal to 1 or -1, meaning the nodes form a linear structure, the node n_i is classified to be part of a dislocation. Otherwise n_i is considered being part of a dislocation junction. The graph collapsing continues using this classification. Using a considerably higher collapse threshold, only edges connecting either two junction nodes or two dislocation nodes may be removed. Apart from some special cases this contraction will eventually reach a stable state, i.e. the speed vectors of all nodes are almost zero. One of the special cases which need to be addressed is shown in Figure 59, where three node form a stable triangle. A similar problem is shown in Figure 60 for four nodes. Both situations are detected by simple heuristics, based on the junction classification, and removed by contracting the whole substructure into a single node.

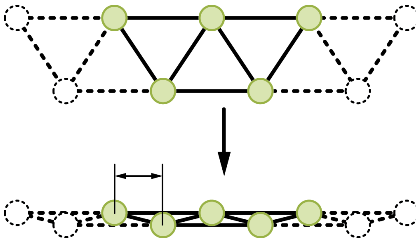

Figure 59: The mass-spring system contracts the neighbourhood graph, but the distance d is too large for the edge-collapsing threshold and the three-edge cycles become stable.

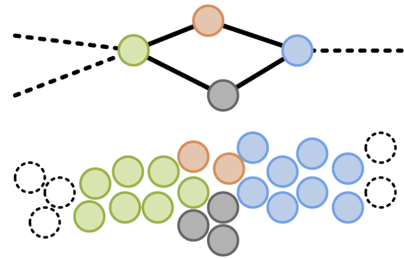

Figure 60: Four nodes and edges in the graph forming a stable diamond which is part of a single dislocation. Yellow and red atoms are incorrectly separated.

After the graph contraction stabilises, a final simplification step is performed, collapsing all edges which either connect two dislocation nodes or two junction nodes without any distance threshold. The result is a graph representing the topology of the dislocation network, as each node now either presents a whole dislocation or a dislocation junction. The junction nodes are placed at the position averaged from the positions of all referenced atoms. The atoms in the sets of the junction nodes are then reassigned to the connected dislocation nodes. The dislocation nodes thus represent a complete segmentation of the atom data. The left image of Figure 61 shows such a result. The final visual representations of dislocations are spline tubes, which are constructed based on the segmented original data (cf. fourth image from left in Figure 57). Junctions are added as spheres, choosing a radius based on the radii of the connected spline tubes.

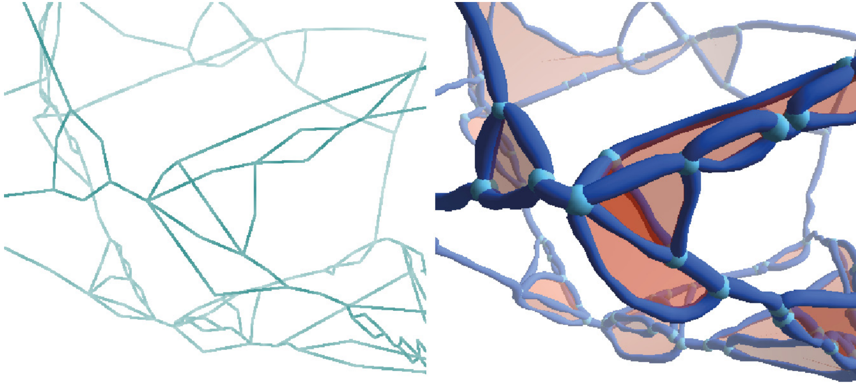

Figure 61: Left: the graph after all simplifications; each edge connects a junction node to a dislocation node. Right: the final result; Junctions are shown as cyan spheres, dislocations as blue spline-tubes, and stacking faults as orange planes.

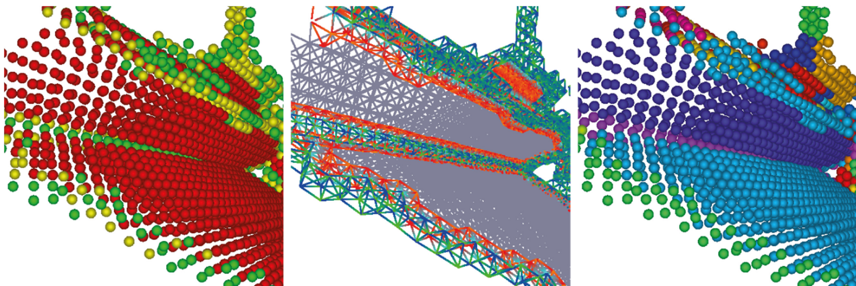

Figure 62: A problematic situation for the stacking fault segmentation using region growing; left: the original atom data; in the neighbourhood graph (middle image) atoms from different stacking fault segments are connected (orange lines between points from both stacking faults visible as grey areas) because of the acute angle under which the two stacking faults meet. The right image shows the final (correct) segmentation.

The extraction of the stacking faults is rather simple, as the bounding dislocations are now known. A region growing in the graph formed by the edges from E_s is performed to identify the individual stacking faults. However, only nodes which have no connection to edges from E_b are collected at first. Figure 62 shows the reason: stacking faults in different crystallographic planes might meet at an acute angle and might thus be connected. Therefore, the first region growing explicitly excludes the border regions. Atoms from these regions are added to the segmentation results in a second region growing step: each atom is assigned the segment ID

which is most common among the neighbouring atoms. This assignment is performed for all atoms in parallel, meaning the originally found stacking fault segments will grow equally fast into the border regions. To get the final visual representation of stacking faults (planes), those dislocations are collected which are connected to the segmented stacking faults. The dislocations are known to form a circle and are thus sorted accordingly. Then a polygon is spanned between the positions of the dislocation tubes (cf. Figure 61, right).

For time-dependent data sets, this structure extraction can be performed for each time frame independently. As all atom segmentations (for dislocations and stacking faults) always reference the original atoms and these atoms have unique id values, it is trivial to correlate atom segments between consecutive time frames and adjust the segment ids accordingly. Some images from a trajectory visualized by the extracted elements are shown in Figure 63. The data set does not change significantly over the first 300 time steps. Then dislocations start to split up and change their form. (E.g. in the lower left area, in front) and large stacking faults emerge (e.g. in the upper right area).

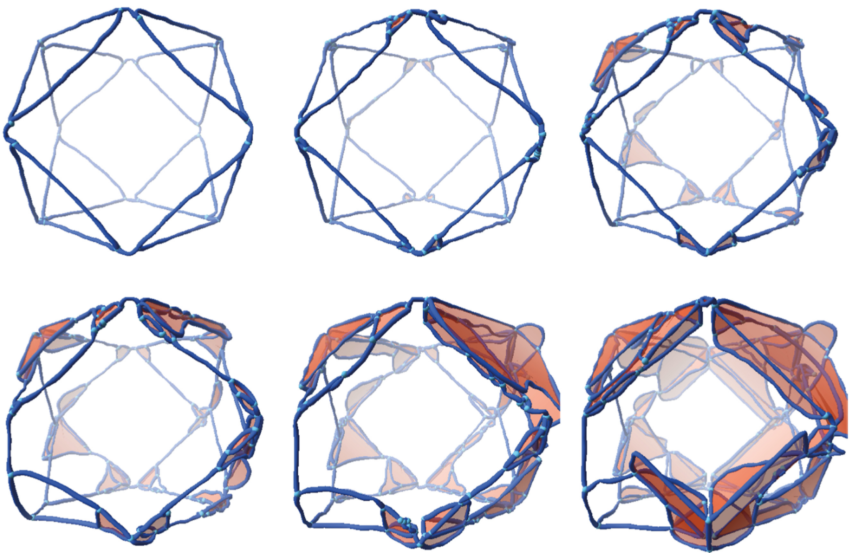

Figure 63: Extracted crystal defects in a time-dependent data set at time frames 10, 250, 350, 400, 500, and 595

The visualization itself is purely polygon-based and the resulting meshes shown in Figure 63 consist of between 28,000 and 93,000 triangles. Such small data is interactively renderable without need of any optimization. The extraction computation itself is more demanding. On an Intel Core2 Duo 6600, 2.40 GHz with 2 GB

RAM these calculation take an average time of about 20 seconds per time frame, depending on the complexity of the structure of the defects. However, this extraction is performed as automatic pre-processing and thus does not hinder interactive visualization of the results.

The presented visualization and extraction is based on several assumptions, which may not always be met. The first assumption is that the undistorted atom lattice is FCC. Figure 64 shows a data set of a compound material of two layers, one being Ni_3Al and one being Ni. Unfortunately, the Ni layer forms a BCC layer in the undistorted case, making the classification, and thus all following computations, fail on this part of the data set. Unfortunately, a similar classification method for BCC-based material is currently not available. To remedy this issue, the classification could be based on the costly evaluation of the Burger-vector. The right image shows clearly that the structure segmentation yields good results for the Ni_3Al layer.

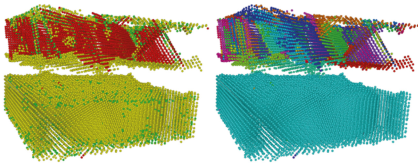

Figure 64: Data set with two layers (upper layer Ni3Al; lower layer Ni) being pulled apart

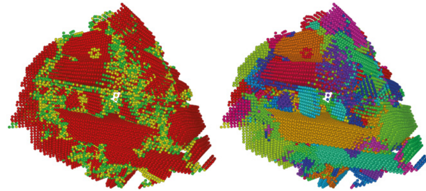

Figure 65: Data set with only Ni3Al pulled in one direction

A second assumption is that stacking faults are planes bordered by dislocations. The data set shown in Figure 65 shows a bulk of Ni_3Al mechanically pulled in one direction. Again, the right image shows that the atom segmentation yields good results. However, the stacking faults are grouped into thick blocks in this data set making the approach of visualizing them with thin polygons unfeasible. Similarly the dislocations do not form a clear network. Following the segment extraction a more generic visualization would be needed e.g. based on the separating surfaces.

3.2 Visualization of Temporal Features

Extracting and showing derived spatial structures is beneficial for the analysis of static snapshots of particle data sets, e.g. for images to be used in printed publications. However, analysing the dynamics of data sets this way is hard, as it usually is reduced to animations of sequences of such images, as has been presented in the previous chapter (cf. Figure 63). This kind of visualization often fails to emphasise complex interactions between elements in the data sets. Tracking particle movement and, more important, structure movement allows to show the dynamics of data sets in still images, e.g. following the concept of path lines. However, tracking

individual particles with path lines is usually not an option, as it results in heavily cluttered images. Additionally, depending on the definition of the spatial structures, the stability over time of these and their tracking is generally not trivial. There is much work on feature extraction and tracking (e.g. [WST+07]), however, best results are usually reached with specific solutions for specific problems. Thus, the generic feature tracking is not discussed in the context of this thesis. Instead, to visualize the dynamics of individual particles several application-focused approaches are presented: two based on path lines, both clustering and combining the information of the whole data set into fewer visual elements which are easier to perceive and understand, and one approach for tracking the dynamics and interaction of extracted spatial structures. The first example shows a generic clustering based on density distribution of the particle data and is thus applicable to a wide range of data sets, although suboptimal for some cases. The second example is a specialized visualization optimized for solvent molecules' path lines in the context of biochemistry. As third example the dynamics and interaction of molecule clusters from the application domain of thermodynamics is presented, which can be applied to other structures in a similar way.

3.2.1 Loose Capacity-Constrained Path Lines

While simulation and quantitative analysis always require the complete, unfiltered data set, visualization and qualitative analysis often suffer from occlusion and clutter for large MD data sets. This is especially true when trying to study the dynamics within such data sets. The core idea is to reduce the number of particles to be tracked to a size which is comprehensible for a human user, while, at the same time, representing the original data set as closely as possible. The reduction of particles will result in a reduction of visible information, but as long as the remaining particles can represent the behaviour of the original data set, this loss of information is insignificant. To not rely on any specific domain knowledge, an approach based on capacity-constrained Voronoi diagrams (CCVDs) as described by Balzer et al. [BH08] can be used. The capacity constraint can be loosened to allow the method to better adapt to non-uniform particle distributions in the input data and to better capture structure and structural changes. This method of loose capacity-constrained Voronoi diagrams (LCCVDs) allows applying the concept of path lines to particle data derived from huge data sets. This approach was originally presented in [FSG+11]. The core idea is to replace the particles by a fixed amount of representatives for tracking and visualization, similar to clustering methods, for which a comprehensive overview is given by Kolatch [Kol01].

Clustering typically groups objects based on a similarity measure such as spatial proximity. A well-known algorithm is k-means [DH73], which assigns points to its nearest cluster centroid and then moves the centroid according to its assigned points, eventually iterating to a stable solution. The result varies depending on the

initial seeds and the number of points assigned to each cluster. Because of the second issue no relation between the density of clusters and the density of the original point set exists. There have been attempts to balance k-means, e.g. [BBD00], [BDW08], [BG02]. However, either the restrictions cannot be guaranteed or they cannot be chosen freely. K-means, as well as the approach of CCVD and LCCVD require the number of cluster centroids as input parameter. While this can be a problem for classical clustering, this is not an issue for the particle reduction task. Quite the contrary, as the presented method aims at a controlled decrease of particles, the number of targeted centroids is the most important input parameter for the LCCVD algorithm.

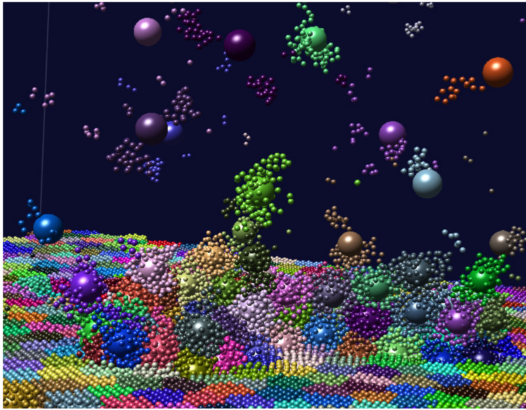

Figure 66: Assignment of points to sites in one time step of the laser ablation data set. Sites are shown as large spheres. The assignment of the original particle data is represented by the colour.

The idea of a CCVD is to create a Voronoi diagram, in which each Voronoi region has a predetermined area, which is called *capacity* of the region's generator point (called *site* in this context). In discrete scenarios, such as particle data, capacity maps directly to the number of particles assigned to a site, which is kept constant in the original work by Balzer et al. Particles are swapped iteratively between sites to reach a centroidal CCVD, i.e. each site coincides with the centroid of its Voronoi region. A result for one time frame of the laser ablation data set can be seen in Figure 66. Due to these iterations the original algorithm is rather slow. Li et al. [LNW+10] presented a multi-core-CPU variant. Balzer et al. [BSD09] utilized CCVDs as optimization method to obtain Voronoi centroids to represent a given density function. However, points in regions of low density are being assigned to centroids comparatively far away as the capacity of nearer centroids may be depleted. K-means clustering does not share this problem, but does not allow drawing

any conclusions about the density of the original data set. Loosening the capacity constraint allows to remedy the effect while maintaining the relation between density of original particles and density of sites. The LCCVDs can be efficiently computed in parallel, e.g. on GPUs using CUDA.

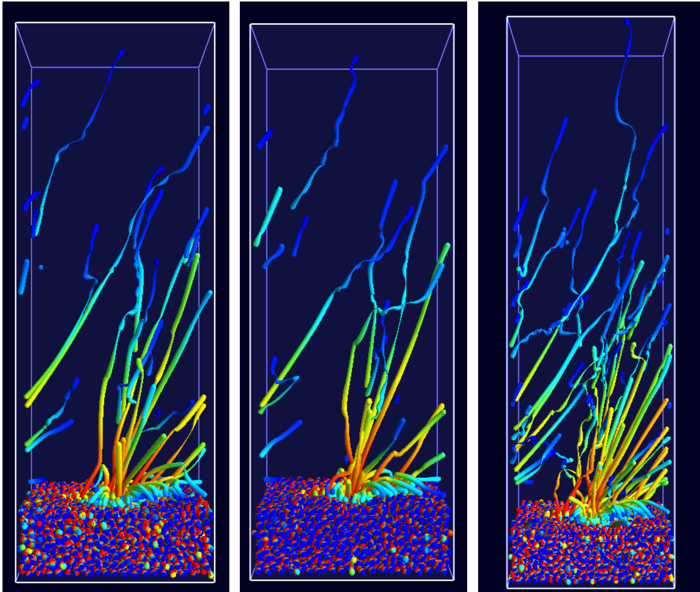

Figure 67: Results for the laser ablation data set (3750 sites for 562,500 points) showing spline tubes as path lines over 400 time steps. Left: capacity constrained to 150, middle: capacity $[100, 200]$, right: capacity $[20, 300]$.

The principal idea of LCCVDs is to replace the constant capacity by a flexible interval: $[c_{min}, c_{max}]$. Each site can hold up to c_{max} points but must hold at least c_{min} points. This introduces the concept of *free slots* for sites, which do not hold points as the site is not at its maximum capacity. The original CCVD algorithm swaps points between sites. LCCVDs additionally allow swapping points with free slots, as long as the capacity intervals are not violated for either site. This capacity interval introduces a new adjustable parameter allowing adapting for different goals for the particle reduction, i.e. reliable representation of density or of movement paths. For the capacity interval $[0, \infty]$ the LCCVD algorithm would basically behave like k-means clustering. For regions of constant density the results typically do not change significantly for different constraint settings. When the density changes or particles are moving fast, the settings have a strong impact on the results. The final results of path lines for LCCVD sites of a trajectory of an MD simulation of laser ablation are

shown in Figure 67, where material is expelled from the dense material bulk into empty space (the data set is also shown in Figure 30). The less strict the capacity constraints are, the more sites are moved out and the further they can be tracked into sparse regions which contain only few molecules. With strict constraints a site might be forced to move towards a more distant particle. In subsequent time frames the site might further move into the new direction as swapping operations might occur with sites following completely different directions. Thus, loose capacity constraints provide a better recreation of regions with lower density and better capture the dynamics within these regions. However, this also leads to a more unsteady site movement and incorrect impression of the density. The capacity interval must therefore be adjusted to each data set. As rule of thumb $c_{max} - c_{min} \approx nm^{-1}$ with m sites for n particles, reaches good results.

The LCCVD creation algorithm was implemented in CUDA allowing utilisation of the massively parallel architecture of GPUs for the computation task. First, in the initialization phase the particle data set is uploaded to the GPU and the sites are initialised, either by placement on randomly chosen points or by reusing the results from a previous time frame. Then, using a kd-tree for an efficient neighbour search, each point is assigned to the closest site which has capacity left. The sites positions are updated to make the sites centroids of the assigned points.

In the main iteration phase points are swapped between sites. The key idea for good parallelization is to restrict swapping to neighbouring sites only. These sites are grouped and point swaps may only happen within such a group. The grouping uses a kd-tree created from the original points. An index is derived for each site depending on the traversal path within the kd-tree to reach the corresponding site. Starting with zero for the root node of the kd-tree, the index is unchanged when descending to the left child and is increased when descending to the right child, by the average number of points covered by the sites in one subtree ($P \times 2^{-l}$ with P being the number of points in the data set and l being the level of the kd-tree; cf. the original publication [FSG+11] for details). Sites close by will thus have similar indices. The sites are then sorted (CUDA radix sort, introduced by Satish et al. [SHG09]) and grouped based on these indices. To allow point swapping with more distant sites, a regrouping occurs such that points can be handed over successively. The regrouping is accomplished by virtually displacing the sites relative to the kd-tree splitting planes and re-performing the enumerations which yields the sorting indices. The displacement directions are the directions of the main axes as well as the corresponding diagonal directions. The displacement magnitude is half of the site group's extent. To limit the number of required displacements and to allow for fast computation of the LCCVD site groups with similar displacement magnitude are handled together. This results in sufficient grouping variation for this implementation to quickly converge to a stable solution.

The swapping operations between sites within a group are optimized using a swapping network. Thus, these operations can be executed in parallel by different GPU threads without the need of atomic operations or locks. Each CUDA thread

performs an LCCVD optimization for a single pair of sites. The sorting network (created on the CPU) determines which pairs of sites of a cluster are handled by a single thread by scheduling the pairs to be processed in parallel. Basically, all pairs of sites have to be considered for this optimization. However, pairs can be excluded if either the distance between two sites is larger than the sum of the respective distances to the point farthest away from its corresponding site, or if both sites have not swapped points for several iterations. The first criterion is beneficial when sites are roughly in their final position but not yet stable. There, the number of sites to test for swapping is typically reduced to six per site on average. The second criterion has a strong impact during the last iterations when many sites already reached stable positions. The whole iteration process is terminated when all sites are considered stable following this definition.

In this implementation, all site groups are processed in parallel on GPU multiprocessors, and all site pairs within a group are processed in parallel in the GPU threads of one multiprocessor. The swapping network ensures each site is only used by one thread and avoids atomic operations. The network is created considering the warp size of the CUDA thread model, allowing all operations to be executed in lock-step without the need for explicit barriers.

To determine which points to swap between a pair of sites Balzer and Heck [BH08] used a max-heap data structure. Li et al. [LNW+10] use the median found by Hoare's quick selection algorithm. Both approaches are impracticable on the GPU. Instead, the worst fitting point can be tracked for each site, by saving its index and weight. Using the criteria defined in [BH08] it must only to be decided whether the swapping operation is carried out or not. This approach converges to the final result slowly, but this is more than compensated by the highly parallel execution.

To compare this CUDA implementation of LCCVD algorithm to the original CCVD method (cf. [BH08], [BSD09]) in terms of performance and quality, a generated 2D point data set was constructed by rejection sampling according to a given density function (constant and non-constant) as input, as the original CCVD implementation is limited to 2D data sets. CCVDs (implementation accompanying [BSD09]) were measured on an Intel Xeon 2.33 GHz. The LCCVD implementation was tested on an Nvidia GTX 285 with a CUDA block size of 128 threads, using 128 sites per site group and processing each site group with 32 threads (warp size). Table 16 details the performance results. For a constant density function the computation of 16384 sites took 106.0 minutes with the original implementation, while the CUDA implementation only required 13.2 minutes. A spectral analysis shows that both methods yield equal quality. The two quality parameters, the normalized radius α [LD08] which should be around 0.75, and the capacity error γ_c [BSD09] which should be close to zero, prove the good quality of the results. The CUDA implementation is significantly faster than the method of Li et al. [LNW+10]. The presented CUDA implementation computes 16386 sites with 256 points each in a constant density setup in 15.6 seconds, while original CCVD implementation requires

35.7 seconds on an Intel Core i7 at 2.67 GHz. As expected, the parallel implementation gets more beneficial the larger the input data set is. For 1024 or less sites the GPU cannot be fully utilized. For 3D data sets the CUDA implementation converges after 45.3 seconds for a task of similar size and complexity. The slowdown is partially due to an increased number of iterations needed and the additional calculations per step. Additionally, the less optimal memory access, i.e. 12 bytes for *float3* instead of 8 bytes for *float2*, presumably also impacts the performance. This could be circumvented using memory padding, which was omitted to be able to handle large data sets.

Table 16: Computation times and quality parameters for varying numbers of sites using the constant and the camel density functions in two dimensions (cf. [FSG+11]). All results were obtained by averaging optimization runs from 10 independently generated sets of sites obtained via rejection sampling. The number of discrete sample points per site was 4096 except where otherwise noted.

Sites	Comp. Time (Seconds)		Norm. Radius α	
constant	[BSD09]	LCCVD	[BSD09]	LCCVD
1024	237.9	165.6	0.7628	0.7576
2048	451.9	204.5	0.7481	0.7482
4096	991.1	252.2	0.7470	0.7445
8192	2413.3	350.4	0.7455	0.7353
16384	6361.8	792.9	0.7367	0.7317
8192 (8192)	8319.1	1427.2	0.7576	0.7473
24576 (1500)	6720.4	304.2	0.7072	0.7035

Sites	Comp. Time (Seconds)		Cap. Error γ_c	
camel density	[BSD09]	LCCVD	[BSD09]	LCCVD
1024	214.6	225.3	0.00349	0.00346
2048	421.9	264.2	0.00291	0.00292
4096	876.6	375.2	0.00263	0.00261
8192	1927.0	626.2	0.00245	0.00240
16384	4911.7	1130.6	0.00239	0.00228
8192 (8192)	6543.7	2562.5	0.00204	0.00199
24576 (1500)	2734.5	234.9	0.00333	0.00318

The performance results for 3D MD data sets are presented in Table 17. The two data sets are also shown in Figure 67 and Figure 68. The first one is a MD simulation of laser ablation of alumina with 562,500 atoms over 400 time steps. The second simulation shows the collision of two liquid droplets (methane and ethane) in vacuum, consisting of 81,672 molecules over 1700 time steps. The results show that the less capacity constrained the sites are, the slower the overall execution is.

This is because of two effects: first, a higher maximum capacity for the sites requires more free slots to be considered for swapping. Second, a very loose capacity constraint increases the possibility for sites to swap points requiring more steps until convergence is achieved.

Table 17: Performance of the CUDA implementation of LCCVDs for different data sets in seconds

Capacity (time limit)	Average	Median	Min	Max
Methane-Ethane Collision, 492 points per site, 166 sites, 1700 time steps				
492 (200 s)	5.1	4.4	0.7	47.5
10-2048 (200 s)	84.8	73.4	25.3	200.0
16-2048	86.9	72.2	24.4	1000.4
350-650 (200 s)	20.3	18.2	7.0	200.0
128-1024 (200 s)	41.0	35.0	14.4	200.0
Laser Ablation, 150 points per site, 3750 sites, 400 time steps				
150	1.5	1.4	0.9	6.4
145-155	5.4	4.9	3.6	40.7
140-160	5.9	5.5	4.0	41.0
125-175	7.0	6.4	4.6	52.3
100-200	7.8	7.2	5.0	58.5
20-300	10.6	9.6	6.8	91.2

Visualising time-dependent data sets by only showing representatives for the sites, results in quite stable motions over time. Sudden changes in the direction only arise in regions with fast, very incoherent movement of the particles. In these cases, sites are forced to swap many points between consecutive time frames instead of following the particles. The path line metaphor is applied to the positions of the sites over time. Time is mapped to colour using cool-warm shading, inspired by annealing, from red to blue. Instead of thin lines, spline tubes allow to use their radius to further convey information. In the visualizations in this chapter the radius is modulated based on the speed of the movement to emphasise discontinuities. Fast movement results in thin splines following the metaphor of a stretched rubber band.

Figure 67 shows the path lines for the laser ablation data set. Please note the PBCs used by the simulation are visible in the visualization as well. All three variants successfully show the sheer movement of the atoms expelled from the solid bulk, which is due to the spatial non-constant application of a laser beam. But the visualization of these free floating particles varies strongly. The spreading of the blast is best shown in the loose capacity constraint case (right image). However, regions with low particle density (in particular in the upper area) are prone to erratic movement, which is also depicted by modulation of the spline thickness. Tightening capacity constraints provide a more steady movement and the sites moving

upwards better adapt to the varying particle density. On the other hand, less sites move away from the solid material which changes the apparent spread angle of the blast. While the middle image preserves the spread angle, the left image shows a very narrow blast of atoms.

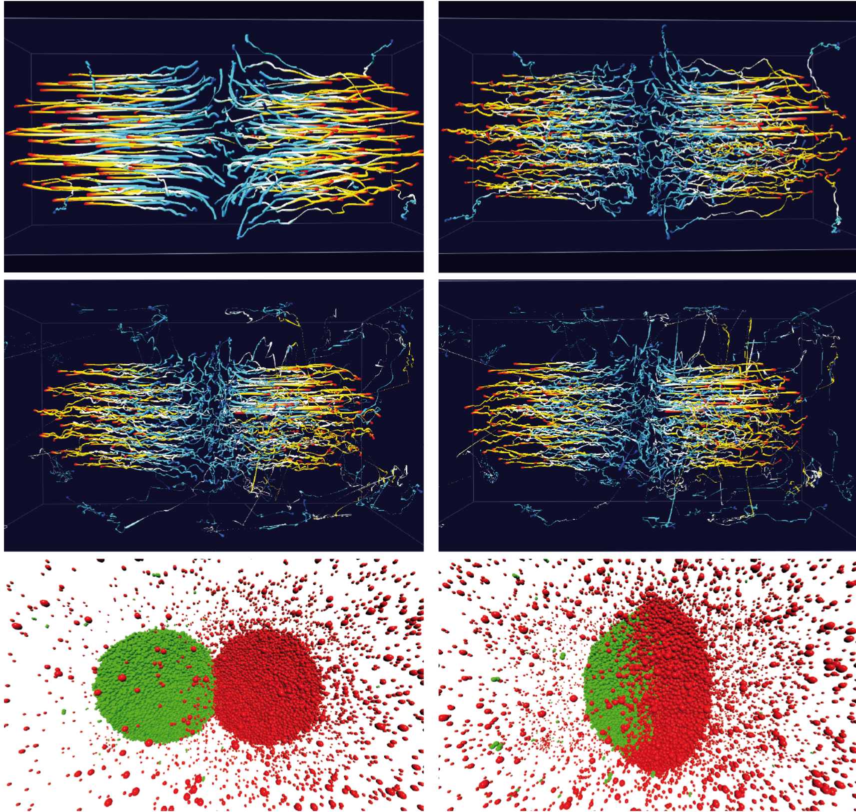

Figure 68: Simulation of a collision of two liquid droplets (methane and ethane) in vacuum for the first 500 time steps. Top four images: path line visualization of sites generated with a time limit of 200 seconds per time frame and the capacity constraints (from top-left to bottom-right) 489 points, $[350, 650]$, and $[16, 2048]$. Right image of middle row: the representation generated with a constraint of $[16, 2048]$ (same as left, middle image) but without the time limit of 200 seconds. Bottom: two snapshots of the simulation: time frames 200 and 500. Ethane is shown in red and methane is shown in green.

The droplet collision data set is shown in Figure 68. At the beginning of the simulation, both droplets have spherical shape. Ethane instantly starts evaporating due to its vapour pressure. When the droplets collide they are forced into an elliptical shape. All variants of the LCCVD visualization capture the initial movement of the droplets as well as the ellipsoidal shape on collision (cyan endings of the splines). In the capacity constraint case (top row, left image) the evaporation of ethane is hardly visible. This effect can best be seen with very loose constraints (middle row). There, however, the methane droplet seems to exhibit a similar behaviour, which is not the case. This is due to the cyclic boundary conditions and the fact that sites do not distinguish between molecule types. In this example, slightly loose constraints (top-right image; capacity constraints $[350, 650]$) best capture the behaviour of the original data.

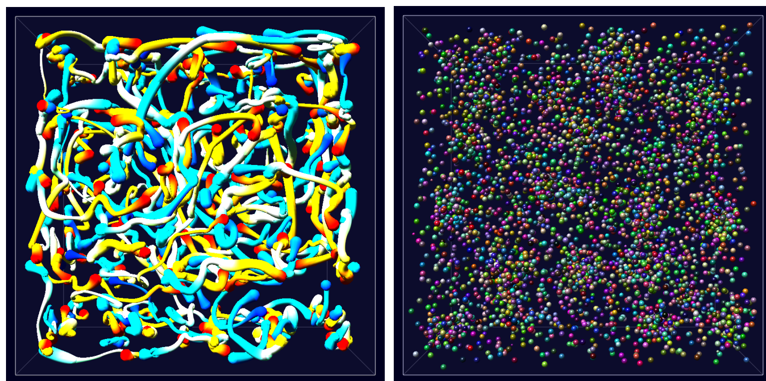

Figure 69: Nucleation simulation of CO_2. Left: the trajectories of the sites resemble homogeneous, chaotic movement (actually present in the vapour phase of the data), but fails to show any clear clustering of the nuclei (constraints $[256, 2048]$). Right: applying domain-knowledge of the data sets and setting the site constraint boundaries low enough to capture the atom cluster size ($[4, 64]$) reveals the atom clusters in the site distribution clearly but results in too many sites for the path line approach.

A third example is shown in Figure 69: a visualization of the nucleation of vapour CO_2 during the transition to liquid phase. The data set is rather small and mainly consists of uniformly distributed molecules representing a homogenous vapour. Local fluctuation in the density results in molecule clusters (cf. Chapter 3.1.1). The spline path lines in Figure 69 (left) fail to capture the formation of the molecule clusters as these clusters are too small (only very few hundreds of molecules) and vanish due to the rather large site capacities. Reducing the capacities Figure 69 (right) reveals these molecule clusters in the site distribution, but results in a number of sites too high for the path line approach. These examples show the benefits

and limitations of LCCVD approach. If possible, an application domain specific reduction or feature extraction and tracking will most likely yield better results. If no such method is available, however, spatial, density-preserving clustering with LCCVDs can be a viable alternative.

3.2.2 Principal Particle Path Lines

An example for an application-specific feature extraction and tracking is the visual analysis of solvent molecules near proteins using principal path lines. Protein-solvent systems are being studied with experimental and theoretical methods to gain insight into the interactions between protein and solvent molecules, in particular water (cf. [PS06], [Hel07]). X-ray crystallography has long been used to analyse water at protein surfaces, since its high resolution provides a detailed picture of protein hydration [Nak04]. These analyses showed that certain positions particularly in the interior of the protein are repeatedly occupied by *conserved water molecules* [BWT06]. The water molecules at these positions interchange within the nanosecond to millisecond time scales, while water molecules at the surface of the protein change in sub-nanosecond time scales [OLW91]. However, directly visualising water molecules' path lines is not feasible as the resulting pictures are cluttered by the unsteady flow of the individual water molecules. A visualization is needed to not only show the conserved water positions, which could be achieved by averaging over time or clustering, but also one that shows the dynamics of the data set with respect to the paths the water molecules travel. An abstract representation of the main routes of water molecules, obtained by clustering their path lines, representing them as tubes, similar to [TvW99], and filtering based on a region of interest was applied to this problem. The approach was originally published in [BGB+08] and [Bid10].

The core idea of the visualization is to extract the relevant segments of the path lines of all solvent atoms and then to construct a network of conserved water positions and principal paths from this data. The extraction of the path line segments is performed as pre-processing step, independent from the visualization. First, a superposition of each frame in the trajectory is calculated, based on a common reference frame, e.g. the first frame. This is computed by a standard analysis tool of the AMBER molecular dynamics package [CCD+05]. The superposition is based solely on the protein's backbone atoms and thus removes the overall translational and rotational motion of the protein while preserving the protein's flexibility. The initial set of path lines is obtained by extracting the solvent molecules which are inside a defined region of interest (e.g. the region close to the active site of the protein) in every time frame of the trajectory. For each of these molecules the positions are extracted which lie inside the region of interest as well as the positions of multiple time frames before entering and after leaving this region (either defined by a number of frames or by a geometric distance). This path line information is

stored as result of the pre-processing and is further processed interactively by the visualization tool.

As the path lines will be shown as static representation of the whole trajectory, the time frame for each position is mapped to colour using a colour map from red, over yellow and cyan, to blue. To further emphasise the direction and speed of motion, a small, repeated, animated saw-tooth luminance gradient is also multiplied to the path lines' colour. As one of the interesting aspects are the conserved water positions, where the solvent molecules move exceptionally slowly, the speed of the movement is also mapped to the saturation of the colour, reducing areas of high velocity to grey colour.

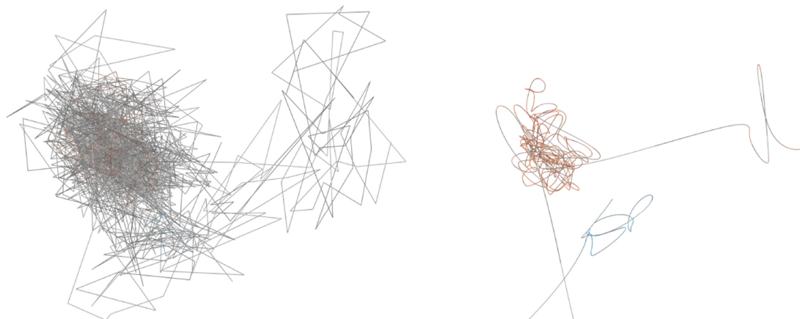

Figure 70: The original path lines are smoothed (right) to emphasise the principal directions of motion normally obscured by the molecule's chaotic small-scale fluctuations (left)

A first processing step aims at removing the high frequency movement from the path line, which originates from the Brownian motion and obscures the principal path. A simple filter kernel ($\cos^2$ in the interval $[0, 0.5\pi]$) is used and the scaling of the kernel is defined by the user. Figure 70 shows the effect on a single path line, as the smoothed version (right image) reveals the principal path, which was hidden in the original data (left image).

Based on this data, adjacent path lines with similar dynamic properties are merged into bundles by a dedicated clustering algorithm. As a first step, the conserved water molecules' positions are detected and clustered. This algorithm is sketched in Figure 71. Positions with low velocity values are selected on each path line independently. Each consecutive sequence of such positions generates a spherical centroid representing this area of slow motion. The velocity threshold must be adapted to the specific data set and the behaviour of the solvent in the corresponding area of interest. The centroid is placed at the averaged position of all selected path line vertices and the radius is chosen such that the sphere encloses 90% of all these vertices. The second step connects the individual path lines by combining the

low velocity centroids of all path lines that overlap with each other (shown as green circles in Figure 71).

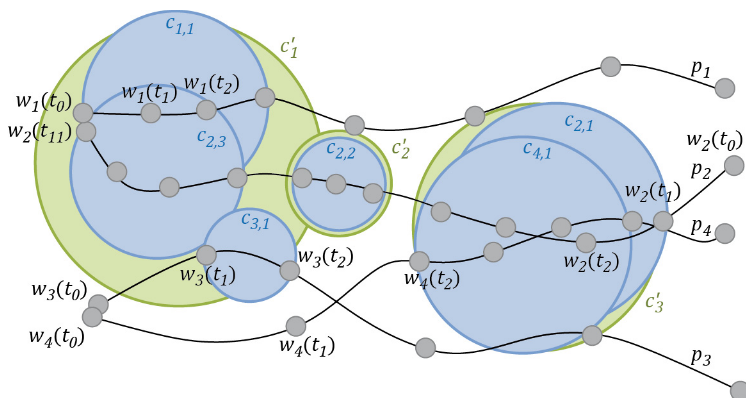

Figure 71: Clustering of the path lines p_i: vertices $w_i(t)$ (grey) along a path line with low velocity values form spherical clusters $c_{i,n}$(marked in light blue). Those clusters overlapping define a new major cluster c'_k (green) connecting the individual path lines.

The centre position of each extracted low-velocity centroid is given by the average of all selected vertices of all path lines which are then grouped. These positions are used as stable locations of the conserved water molecules. In addition helper centroids are placed at open ends of the path lines to connect the network to be extracted with the bulk solvent. The segments of all path lines which consist of vertices with velocities higher than the corresponding threshold are now extracted as connections between the stable positions. All such vertices of all path lines connecting the same pair of stable positions (in the same direction) are collected to form a single edge. The only additional value extracted is the number of original path line segments these vertices originate from. These edges thus represent a bundle of similar path line segments. A simple spline tube is a fitting visual representation to show the path as well as the number of grouped path lines. As the line segments are rather short, a single cubic Bézier curve is sufficient. It is fitted into the selected vertices and the thickness of the tube is derived from the number of path line segments collected. A colour code (from yellow to blue) is used to encode the direction of motion of the original path lines onto the tube (cf. Figure 73). The main problem that remains is that these segments usually highly overlap with each other, especially the two edges connecting a pair of stable positions in the two opposing directions. To remedy this issue, the inner control points of the Bézier curves are adjusted to slightly separate two such curves. Depending on their distance, all other curves have a slight repelling influence on these control points. The

details can be found in the original publication [BGB+08]. The result of this adjustment can be seen in Figure 72. The main directions of the curves are not altered but the issue of overlapping of tubes is removed.

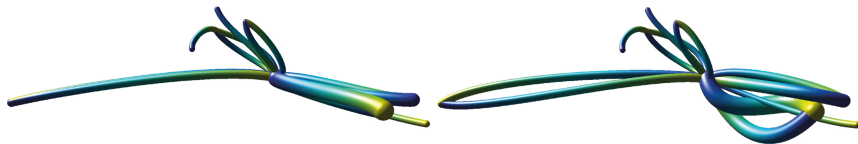

Figure 72: A small subset of a principal path line network before (left) and after (right) adjusting the Bézier curve control points based on neighbouring curves (image from [Bid10])

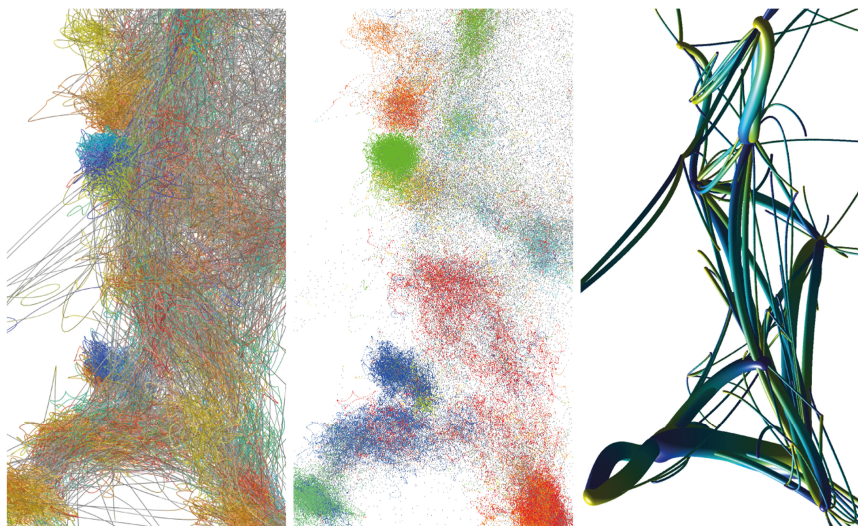

Figure 73: Extraction of principal path lines network. Left: all path lines after smoothing; Centre: the vertices of all path lines, coloured based on the detected clusters of slower motion. Positions of conserved water found through crystallography correspond to the larger clusters. Right: the extracted principal paths

In the original publication [BGB+08] these extractions and visualizations were used to study and compare two variants of TEM β-lactamase proteins, which play a major role in antibiotic resistance of gram-negative bacteria. Two data sets with approximately 25,000 water molecules and 50 ns trajectories (i.e. 50,000 time steps) were processed. One of the data sets is shown in Figure 74 after extracting

and heavily filtering the principal path line network. The visualization confirmed the previous assumption that water molecules can enter a cavity at the active site of the protein through two distinct channels, of which only one is clearly visible in the data from X-ray crystallography. The additional route (on the lower left side of the image) proceeds through a dynamically formed opening in the middle of the red protein loop. The existence of the second route could be assumed based on classical path line representations, but the principal paths' network helps understand the actual motion of the water molecules and the function of the protein loop.

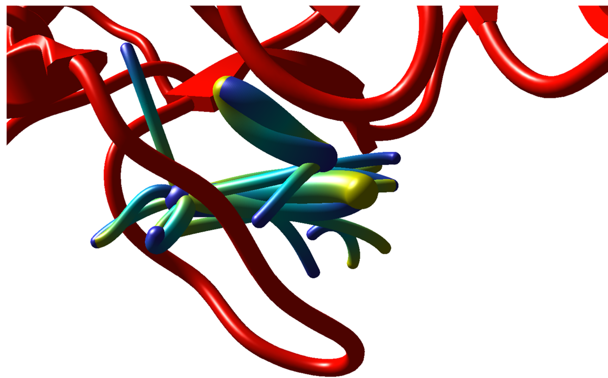

Figure 74: Structure of principal solvent paths in context of a cartoon representation of a surrounding protein (red). The image shows the extracted main solvent paths. The time frame is mapped to a colour gradient from yellow (start of the trajectory) to blue (end of the trajectory).

The positions of slow motion detected by the presented extraction agreed with the known locations of conserved water positions. The tubes representing the main paths of solvent molecules help to follow and comprehend the movement of water molecules between conserved positions inside the cavity. It becomes evident that there is no straight route for a water molecule once it has entered the cavity, rather is seems that water molecules quickly interchange between their relatively long stays at the conserved water bridge positions. Another interesting fact is that the motion of solvent molecules inside the cavity is rather structured, compared to the chaotic movements in the bulk solvent area.

Just recently [LBH11] presented a similar approach for analysing the docking of proteins. They extract the possible paths to an active site of a protein a large molecule can take. The analysis is based on Voronoi diagrams. The resulting path network is presented in a way similar to the approach presented in this chapter.

3.2.3 Dynamic Interaction between Molecule Clusters

Both of the previously shown methods visualised the dynamics of data sets in the spatial context of the original particle data. On one hand this is beneficial because the derived representations are easy to interpret because of their direct relation to the known physical space. On the other hand, however, this limits the possibilities for visualization and the amount of elements to be represented, since otherwise the resulting images would suffer from overload, occlusion, and clutter.

Thus, an alternative approach for static visualization of the dynamics of MD data sets is to create abstract, often diagram-like representations, decoupled from the original 3D space. In such diagrams the behaviour of specific features of the data set can be shown in an optimised context. The system presented in [LBM+06] follows this approach and visualises turbulences of the mixing layer between two fluids of different densities. Their application presents a three dimensional rendering of surfaces of the mixing layer in combination with a schematic view of features (bubbles) of the mixing turbulences, which they refer to as merge-split graph, visualizing important events in the features' evolution like births, merges, splits, and deaths. This chapter presents one example of how this concept can be applied to the tracking of interaction between molecule clusters in the context of nucleation processes in thermodynamics (cf. e.g. Chapter 3.1.1). This application was first presented as part of [GRVE07]. The main goal is to judge the applied cluster detection criterion, in terms of the correctness of the behaviour in relation to the knowledge gathered from experiments and classical theories. As quantitative measurements are best automatically calculated, the visualization needs to address qualitative features, like the stability of clusters and their interactions with each other and with the surrounding bulk vapour as the clusters change their size. For example, comparing the results of two clustering criteria on the same data set is hardly possible with direct visualization in the 3D physical space.

First, a definition and selection of the features to be included in such an abstract visualization is needed. An obvious choice is to use the molecule clusters, as their behaviour over time is in the focus of this application. This also requires *time* to be a distinct element of the representation, e.g. the value for one of the axes. The attribute of the molecule clusters which is of most interest is their size in terms of agglomerated molecules. The aspect omitted is where these molecules come from, in case the cluster grows, or where the molecules go to, in case the cluster shrinks. The more interesting aspect is the interaction of clusters with each other. Cluster at close proximity might influence each other and might even coalesce. Thus the visualization focuses on this feature. Therefore, *flow groups* can be defined to characterise the interchanging flow of molecules between clusters. Assuming each cluster has a unique ID value, each flow group is then defined as the number of molecules leaving a cluster e at time frame t_e and joining cluster v at time frame t_v. The flow groups must be computed in a pre-processing step as data has to be gathered from the whole trajectory. However, it is not meaningful to show all flow groups. Two

filtering steps should be performed to remove unnecessary data. First, flow groups which contain too few molecules can be omitted, as their relevance is too low compared to the risk of cluttering the image. Second, flow groups traveling over a time too long should also be removed, as these molecules more likely just evaporated into the bulk and only by chance joined a second cluster at a later time step. These two aspects are not independent, however. Flow groups existing only for a very short time might be of interest even if they only contain very few molecules, only a single molecule in the extreme case. On the other hand, flow groups existing for a long time and consisting of very many molecules may indicate a directed, structured behaviour or a severe malfunction in the cluster detection criterion and should therefore not be removed.

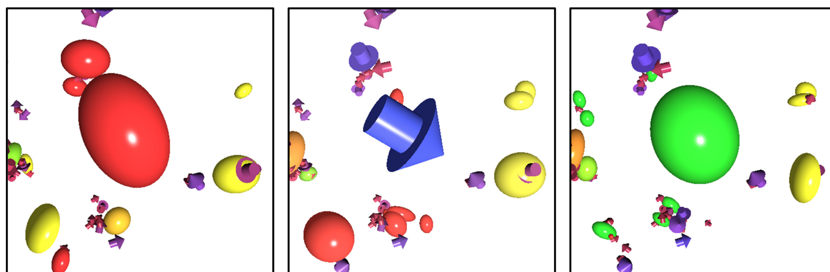

Figure 75: Clusters with colours between yellow and red are decaying (left), while clusters with colours between yellow and green are growing (right). Evaporating molecular cluster (in the middle of the left image) leaving a flow group (blue arrow, centre image) containing almost all molecules of the evaporated cluster (indicated by the similar size), moving slowly and forming a new cluster (right image). This behaviour hints at a cluster detection problem.

Molecule clusters and flow groups can be shown in the 3D space (cf. Figure 75), providing an easy interpretation, or in an abstract diagram (cf. Figure 76). Of course, both representations can be combined by coordinated views providing linking and brushing, e.g. for selection of clusters and corresponding filtering. Using such a tool a user can easily get more information about the detected clusters, their evolution over time and their interaction with each other and with the surrounding monomers.

In the 3D space representation, molecule clusters can simply be shown as ellipsoids (cf. Chapter 3.1.1). To emphasise the evolution, a special colour coding can be used. Three user-defined colours represent the major evolution tendencies of clusters. The default colours are yellow for clusters keeping their size, green for growing and red for decaying clusters. These colours are interpolated accordingly: e.g. slightly shrinking clusters are coloured in orange. The surrounding vapour can be shown by different metaphors including point-based GPU ray cast spheres. Details can be

found in the original publication [GRVE07]. For easier viewing of the elements of interest, i.e. clusters' ellipsoids and flow groups, all remaining molecules can be filtered. The flow groups can be shown using arrows glyphs (cf. Chapter 2.5 for GPU ray casting). As these arrows are based on many individual molecules, their movement is averaged and abstracted: the flow groups' arrows move linearly from their emerging position, averaged over the position of all atoms at the time frame t_e, to their vanishing position, averaged from all atoms at time frame t_v. The sizes of the arrows are given by the number of molecules the flow groups contain, scaled accordingly (based on cubic-root and considering HCP) making them visually comparable to the ellipsoids. The remaining parameters of the arrow, length and radii of arrow head and tail, are chosen relative to the main size value, such that the overall impression corresponds to the sizes and impressions of the molecule cluster ellipsoids. Figure 75 shows an exemplary flow group connecting two clusters. All three elements are of almost equal size, which means that they are likely to all represent the same set of molecules. This indicates a cluster detection failure, but can also be because of normal density fluctuations, because all elements are rather small. To further increase the information coupling between the arrows and the ellipsoids, the colour coding of the arrows is chosen to represent the fraction of molecules in the flow group and in the cluster. A value of 1, meaning all molecules of the cluster join into the same flow group, is represented by blue (which is not used by the colour coding for the ellipsoids). The value of zero is shown by a red colour. Small values close to zero mean that only very few molecules of the cluster join in the corresponding flow group. The colours are chosen in relation to the clusters e and v and are interpolated as the flow group moves. The flow group shown in Figure 75 being clear blue further emphasises the fact that it contains all atoms of both corresponding clusters.

The schematic view of the dynamics of the data set is shown in Figure 76. The horizontal axis represents time. Molecule clusters are represented as horizontal lines of varying thickness, encoding the number of contained molecules. The vertical arrangement of the clusters is initially based on the cluster IDs, but can be manually adjusted by the user. The clusters do not have a further colour coding and are shown black, with the only exception that selected clusters are shown in red. Selected clusters in the 3D visualization are rendered with a halo, as colour modulation does not provide enough distinction. Molecule cluster lines can be selected by either clicking on the line in the schematic view or by picking the corresponding ellipsoid. Flow groups are shown by Bézier curves. Their start and end points are given by their clusters and the corresponding time frames. The thickness is derived from the number of molecules using the same mapping as for the lines representing the clusters themselves. So the amount of molecules leaving the cluster at a time, represented by the decrease in the line width, can be compared to the amount of molecules contained in a flow group. The schematic view can be interactively zoomed and panned and can be filtered to only show the selected cluster lines, the

connected flow groups, and the cluster lines directly connected to the shown flow groups.

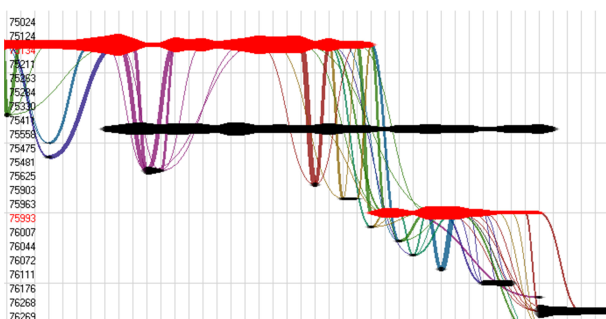

Figure 76: The schematic view of the molecule cluster evolution with two selected clusters (red) and all corresponding flow groups connecting to other clusters (black) shows a 10,000 methane nucleation dataset with geometrical cluster detection. The colour of the flow groups are based on the IDs e and v of the clusters they are leaving and joining to allow for visual grouping.

The fluctuation in molecule clusters, e.g. shown in Figure 75, also manifests for larger clusters, which are always surrounded by small flow groups. This effect is partially expected as molecules may collide rather fast with these clusters, but cannot join them immediately because of too different speeds and energy levels. Instead, they rebound, get slowed down, and then join the cluster some time steps later. There are different cluster detection criteria, which handle effects like this differently. As a matter of fact there is no thermodynamically correct detection algorithm. Each method is an approximation of quantum mechanics and has advantages and disadvantages. Although in the application domain there exists detailed knowledge of the limitations of these methods, comparing results of different detection criteria on the same data set is important to judge the quality of the found molecule clusters and the derived values, e.g. nucleation rate and critical cluster size.

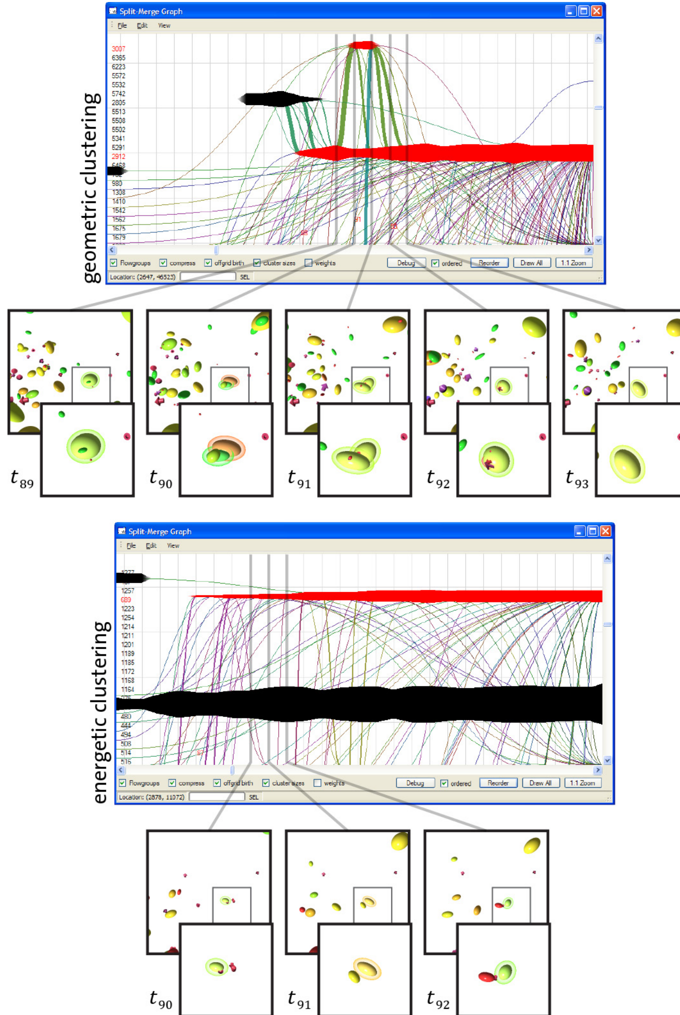

Figure 77: Schematic views of two clustering algorithms applied to a 50,000 methane nucleation dataset. The top images show the results of a pure geometric clustering, and the lower images show the results of a clustering based on energy levels. The images below the diagrams show zoomed-in views of the selected cluster (marked red in the diagrams) at different time frames. The geometrical clustering detects large and unstable clusters. The selected cluster is temporarily split into two. The thick green flow groups and the corresponding dent in the lower cluster line indicate a detection problem. The energy-based clustering (right) is more stable in this respect but results in smaller clusters.

Figure 77 shows the schematic views for a methane nucleation simulation data set with 50,000 molecules comparing two cluster detection algorithms. The top images are generated based on a simple geometric distance criterion. A molecule with four neighbours within a distance threshold is considered to be part of the liquid phase. All molecules within this neighbourhood and which are in liquid phase form a cluster. The algorithm used for the bottom images defines two molecules as clustered if the sum of their potential energy and their relative kinetic energy is negative (cf. [Hil55]). The geometrical criterion detects larger and more clusters than the energy-based approach does (cf. Figure 77 in which the same cluster is selected, which is number 2912 if using the geometrical criterion and number 689 if using the energy-based criterion). It also often creates multiple clusters, close together, instead of a single one. The small cluster at the top of the top image in Figure 77 (ID 3007) is such an example. The line of this cluster is connected to the lower and bigger cluster (ID 2912) with thick flow groups, indicating that almost all molecules came from and re-join that cluster. At the same time (time frame 91) the big cluster (2912) shows a dent in its size corresponding to the size of the small cluster (3007). The energy-based cluster detection shown in the bottom diagram does not exhibit such splitting.

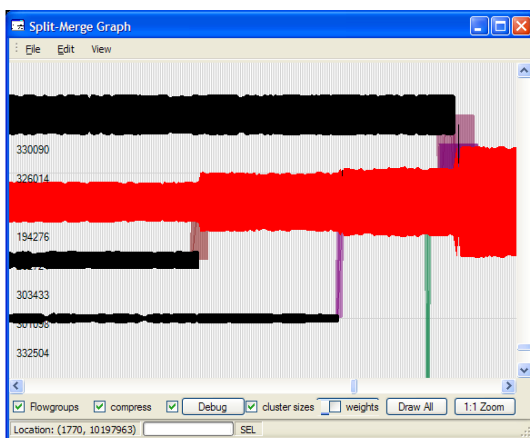

Figure 78: Schematic view of a R-152a nucleation simulation (Figure 4; right). Almost all molecules are very quickly clustered in many rather small and stable clusters. Greater changes only happen when two clusters merge, which is clearly indicated by the corresponding thick flow groups in the schematic view.

A second, interesting observation in the top image of Figure 77 is the black cluster (ID 2805) which disappears and feeds all molecules with several flow groups to the big, red cluster (ID 2912) over a time of about four time steps before it com-

pletely vanishes. The coupling with the direct 3D view revealed the simple reason behind this behaviour: The cluster (ID 2805) became rather slim and elongated at time frame 88, such that the geometrical method failed to get the required four neighbours for some molecules to cleanly detect the cluster. Instead, two clusters are detected. When the cluster changed back into a more elliptical shape, the detection criterion did not re-assign all molecules to the original cluster but to the new one. A similar effect can be seen in Figure 76 between the two selected, red clusters.

The schematic view does successfully represent the behaviour of the data set. A further example is shown in Figure 78. This data set, also shown in the right image of Figure 4, is a nucleation simulation of the cooling agent Difluoroethane (R-152a). Early in the simulation, most molecules cluster in rather small droplets of only few molecules each. Later on, almost no fluctuations between the clusters take place, but clusters coalesce with each other. This is clearly visible in the schematic view by flow groups at the end of the cluster lines, which are of same size as the clusters themselves and which connect to a larger cluster within a very small period of time (usually only a single time step).

This exemplary, abstract visualization of the dynamic of MD data sets can be extended to other types of data and other application fields. However, what becomes evident is that a generic visualization cannot be optimal in all cases. Instead, visualizations optimized for specific problems and designed and developed in close collaboration between visualization experts and research partners from the corresponding application domains will usually yield better results.

4 MegaMol

To recapitulate one important statement of the introduction: The purpose of visualization is to gain insight into large data sets from diverse applications (cf. [CMS99], [JH04], and Chapter 1). As such, visualization needs, apart from presenting some novelty, to work with real-world data sets and needs to be applied to real-world problems to advance the scientific field itself. To be able to handle the continuously growing data sets and problem sizes the software complexity also increases to achieve the required solutions. Software systems emerge to cope with this situation and are quickly gaining importance, which can be seen in the fact that major visualization venues recently introduced the *system paper* type. However, such software systems tend to be huge, continuously growing, and more long-lasting than originally intended and expected [Phi98]. Creating such a large software system benefits from expertise beyond the field of visualization itself: namely from a deeper understanding of software design and the principles of software engineering (SE). In this chapter this concept is detailed by the example of *MegaMol [Meg]*, a visualization system focused on particle-based visualization for MD data sets. Almost all visualizations presented in this thesis were implemented as part of MegaMol.

4.1 Visualization System Engineering

However, the increased data set sizes alone are hardly a believable reason to start developing software systems, as scientists have worked with large data sets before. The main issue is given by the fact that the problems from the application domain which should be approached with visualization themselves also increase, in terms of complexity. Users want to employ visualization as flexible and usable tool for their analysis process. Otherwise, the software is reduced to a generator for images for scientific publications. However, as a tool for analysis tasks, a visualization system must exhibit high quality in terms of flexibility, stability, and usability. For example, users from the application domain, which will most likely not be computer scientists, must be able to adapt the visualization workflow within the system to their specific and current needs, which, in turn, are likely to change during their analysis. Such a flexible software system is complex and large and requires extensive efforts to be set up. This effort needs to return in terms of advantages for future research. Otherwise, such software cannot be implemented economically, in terms of financial development cost and time. In the academia it is highly unlikely that this initial effort can be made, because researchers need to quickly switch between different projects and long term funding is usually coupled with task and problem lists extensive enough to fully occupy the corresponding researchers. Given the limited available time [KKEM10] the resulting implementations will usually not go beyond the mere technique and will cut some corners, e.g. hard-coded parameters and simple data file formats. There will be no carefully crafted architecture or thor-

ough error handling as such an implementation is enough to produce performance results as well as few screen shots to add to the publication. Especially with at-the-edge concepts like assembler shaders, GPGPU APIs, and the constantly evolving graphics card drivers and bugs, researchers often are happy enough that the code works at all. And, as it is not a task for researchers to write production software to generate revenue, delivering proof of concept with disposable prototypes seems sufficient.

To cope with the increasing problems, however, disposable proof-of-concept prototypes are not acceptable. Suitable software systems will exhibit the attributes of production software, like sophisticated file formats and memory management to cope with large datasets of different format and from inhomogeneous sources, end-user-friendly interface for the parameterization, and robustness against software and operating errors, which seems a contradiction to the priorities of a research environment. The situation is most pressing in the visual analytics community, where, by design, a large number of publications are systems-centred since the visual analytics workflow itself utilizes techniques from many different fields to perform complex analysis tasks. This is duly recognized as a significant challenge with no immediate solution in [KKEM10]. But also for the visualization community this issue is gaining importance. There are many books on data structures and optimization at the algorithmic level for visualization [GP07], [KKEM10], and utilising SE for the field is probably just a next step. Cleanly Designing and implementing software is by no means impossible. It usually corresponds to the effort for application papers, where the focus is partially shifted from the core algorithm to, e.g. the user interface. Systems require significantly more time than writing a disposable prototype, as each of their different parts or modules already have roughly the same complexity of implementation as a single disposable prototype, but need to be more generic, which adds some further overhead [VHS01]. The question arises how to efficiently use the available working resources to tackle this problem.

Employing existing programs or modules other researchers offer as public domain (implying due acknowledgement) seems an acceptable solution. Integrating research modules into existing systems might actually be the best solution, if applicable. Each system, however, was designed with a specific goal in mind and an integration of research code will only succeed if this coincides with the required research direction. AVS [AVS] pioneered the field of extensible, generic visualization software systems, as it features a thin framework, allowing for the interactive composition of functional modules to define the resulting application. However, it only offers low utility functionality and it is not actively maintained, thus lacking modern rendering techniques. VTK [SML97] offers many classes for data types and algorithms, often with multiple CPU-based implementations. As the main goal of its developers is general applicability optimizations for special cases or special hardware (even GPUs) are mostly omitted. ParaView [JAL05] is the most prominent reconfigurable frontend based on VTK. It offers good performance through customized modules and is ready for distributed and parallel rendering in cluster environ-

ments. Representatives for generic frameworks with focus on information visualization are e.g. Prefuse [HCL05] and Improvise [Wea04]. They are extremely flexible and offer sophisticated data management including object sharing, events for data manipulation and user interaction, and garbage collection, implemented through intrinsic features of Java. These tools include a wide range of important functions required like data query languages, graph layouting support, flexible data tables as well as most of the commonly used visualizations. Writing additional computation or visualization modules is relatively easy for all of these systems once the user gets acquainted with the corresponding architecture. However, if a certain concept of the system, like a data handling paradigm or available data types conflict with the requirements of the researcher the benefits from the framework's functionality are extremely limited. A secondary framework will most likely be required to be grafted onto the existing one, which results in almost the same effort as writing everything from scratch. This is the case for particle-based visualization for large MD data sets. Existing frameworks often focus on continuous grid-based or mesh-based data and almost always lack support for large particle-based data, e.g. linear block storage for fast rendering or streaming capabilities. Available open-source tools especially written for MD data or other particle data focus on specific problems, e.g. protein visualization, and are thus too inflexible (e.g. VMD builds up internal data structures for protein analysis tasks, which hamper high performance rendering of very large particle data sets, cf. Chapter 2.3). To meet the requirements for the large research project funded through the SFB 716 [DFG] the MegaMol project was started with the goal to visualise data sets with 10^7 particles from physics, material science, biochemistry, thermodynamics, and engineering. There was a program available at the VIS[8] research group for point-based visualization of galaxy data sets [HE03], which was extended to handle MD data [GRVE07]. Being a classical research prototype, the program had an extremely monolithic structure and was yet extended several times. It had reached a state where maintaining its full functionality was not economically acceptable any more.

But, if there is the necessity to start a new software system at least some publicly available components could be used to lower the implementation workload. While this is true to a certain extent, a system which was quickly put together by only existing components, e.g. as it is common practice in web programming, exhibits similar problems as a disposable prototype: while the functionality, the usability, and perhaps even the flexibility required to be used as a system might be reached, the lack of an adequate software architecture is bound to decrease the scalability in terms of performance, as well as the stability in terms of robustness, extensibility, and maintenance of the software itself.

Scalability in terms of rendering performance and data sets size is achieved, among other factors, by the scalability of the employed algorithms and the imple-

---

[8] Institute for Visualization and Interactive Systems; University of Stuttgart; Germany; http://www.vis.uni-stuttgart.de/

mented memory management. Performance of the algorithms can be ensured even in a composed application, as it mainly is an issue of optimizing the individual modules internally. In contrast, optimizing the in-core storage must be accomplished on a larger scale. Issues arise from the fact that the different modules might require different data layouts and may even be written in different programming languages. The latter aspect is not a problem by itself, at most a conceptual challenge, but might result in the need for data marshalling between the different language paradigms which, again, can introduce overhead. In addition, many freely available implementations favour educational usefulness and thus simplicity of code instead of efficiency or scalability, and many algorithms are implemented in Java because of the popularity of the language, resulting in the well-known scalability issues and memory restrictions of un-optimized implementations. For example, the developers of Prefuse explicitly mention a performance bottleneck for visualizations of even medium-sized data sets with a few thousand items[9].

A more severe problem, however, is given by the need for different data layouts. This often results in data replication, which in turn, even assuming every component scales well in terms of performance, increases the required in-core memory with every additional module. Only a centralised base architecture of a shared data handling can overcome this problem. This, however, requires either extensive work on the framework, e.g. providing facade access interfaces for all required modules and programming languages, or requires adaption of the originally unchanged, incorporated code modules.

Multiple imported modules, which must be considered semi-black boxes in a composed system, also introduce the problem of maintenance stability. When the different modules communicate with each other, a common case e.g. for coordinated views or multi-pass algorithms (with different passes being realised by separate modules), the number of code paths between the modules calling each other inevitable grows. This interconnection is referred to as coupling [PJ88] and is an essential aspect of software quality. There are five different levels of coupling, but only the worst two need to be avoided to obtain maintainable software. One is *content coupling*, where modules can address internal data of other modules, which endangers the validity and consistency of the data. The other one is *global coupling*, referring to global data storage without explicit data ownership. Therefore, every access potentially interferes with every module. The other three coupling variants influence internal control flow (*control coupling*), pass complex data types (*stamp coupling*) or just elementary data types (*data coupling*) between modules. It is usually not feasible to guarantee this minimum coupling in research prototypes, as the interfaces for all modules would be needed to be known beforehand. Instead, modules communicate with each other in arbitrary fashion though interfaces written and extended on demand, in the worst case resulting in content coupling. Making changes to one module or adding a new module therefore has negative side effects

[9] http://prefuse.org/doc/faq/#tec.1 (last visited 29.12.2011)

on many other parts of the system, rapidly decreasing the maintainability and robustness of the whole software, which will thus eventually break down completely.

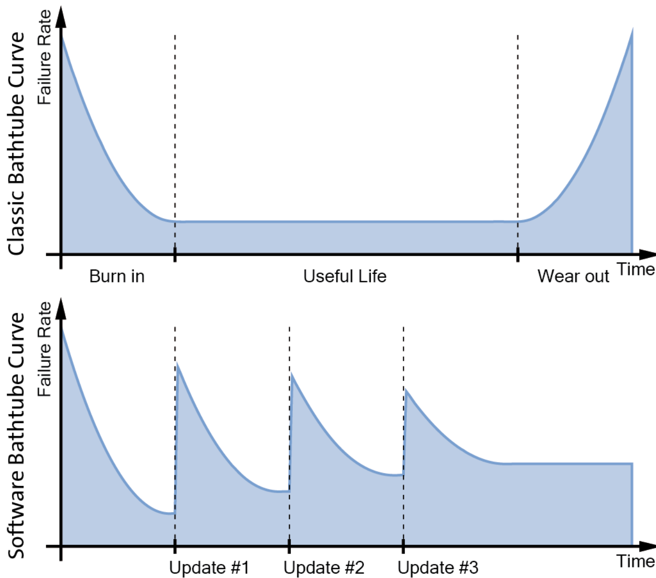

Figure 79: Top: the classic bathtub curve as known from traditional reliability analysis. The integral of the curve until a given point in time relates to the effort invested. Bottom: the bathtub curve for software [Rel96] exhibits two main differences: software does not decay and thus cannot wear out. However, it will be updated several times over its useful life, which will always add new bugs. In the long run, functionality will not fit the original design and thus deteriorate quality, increasing the failure probability with each update.

For the formalization of this scenario, the *bathtub* curve, known from traditional reliability analysis [Rel96], can be adapted to the life cycle of software (cf. Figure 79), which is not prone to physical fatigue, but which gets less and less maintainable with each update introducing a significant amount of new functionality. The increasing complexity of the software results in an increasing asymptote for the reliability, i.e. the software quality steadily decays over time. The diagram also reflects the effort invested into the development process as the integral of the curve up to a given point in time. The authors of the software-adapted curve suggest improved SE as the only way to counter this effect and to reduce failure rates again. The question arises if this is true for all types of software, or only for systems.

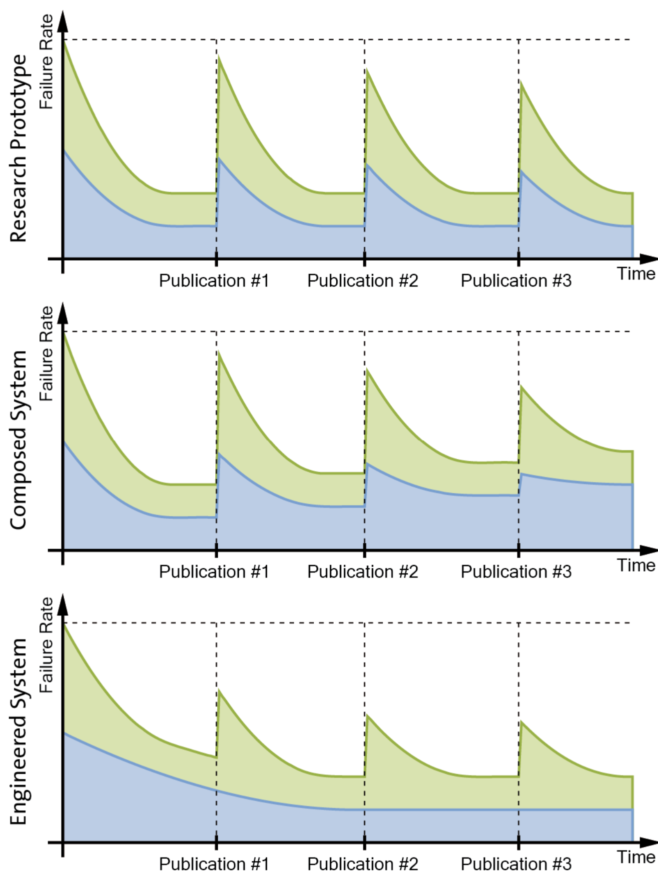

Figure 80: Bathtub curves for different kinds of software developed in the context of research. The blue curve represents the base workload required for development of framework aspects, e.g. data set loading or the main loop. The green curve (identical in all three diagrams) adds the workload required for the algorithmic method development for each publication.

The software bathtub curve can be further adapted to represent the features of different types of software. These curves are shown in Figure 80. The top-most curve models the characteristics of *proof-of-concept prototypes*, which exhibit a one-time development effort, ideally resulting in a publication. Only the core algorithm is optimized for performance. The whole process basically is repeated for each publication anew, although some code elements can be copied and the new work bene-

fits from lessons learned, resulting in slightly less cost. Through reuse of code, features and bug fixes accumulate, but at the same time code will be reused for tasks, which can be made to possible, but which the code was never intended for. This results in a reduction of software quality. If the resulting pseudo-system reaches a state where it is no longer usable, e.g. because of the missing architecture adding new features is not possible at acceptable cost, much of the code base is removed and restarted from scratch. Because of this, research prototypes exhibit a nearly constant effort and nearly stable reliability. From an SE point-of-view, research software for application papers is just a variant of this, where part of the effort is shifted towards a viable user interface and increased robustness.

The centre diagram depicts the situation for *systems composed* from freely available components. The workload required for the system decreases for each publication, as later publications can utilize the existing framework functions. However, these systems are started with a specific goal in mind, usually for a specific first publication. Further work will change this goal as research questions change over time. However, just because the system is available, it will be used whenever possible. As the development time is going to be reduced because of the well-known time constraints there will be no thoroughly designed architecture. Each time new functionality is added via a new module, either newly written or imported from existing code, it needs to be connected to the existing system. The lack of interfaces and the need for quick results will result in code replication, redundancies, and, in the worst case, content coupling [PJ88]. The manipulation of internal data of most modules and ubiquitous side effects will thus deteriorate the stability of the framework and will increase the failure rate, resulting in increased maintenance effort, as shown in the original software bathtub curve (bottom image of Figure 79). The system will eventually reach a point of instability where the architecture will break down and the maintenance cost will increase unacceptably.

Since a composed system has basically similar issues like proof-of-concept prototypes, it might also have the same justification of being sufficient for a single or only few publications, rendering the overhead of SE for visualization systems superfluous. This, however, is not true. On the one hand, the required effort to set up a system, even just a system composed from existing code, is too high to be justified for a single or very few publications. On the other hand, the sheer availability of a system of any kind results in the desire to employ it for further applications. The life span of the software thus increases beyond to original plan, which results in tweaking of modules towards changing goals. The system's structure, however, will change only very little or not at all [AR99]. As a result of the issue that the framework was not designed properly, the structure will develop into a hindrance in the long run.

These effects of framework degeneration are countered by a well-thought-out design created before implementation starts, which yields stable frameworks of *engineered systems*. These will also be extended for each publication, but because of the initial design effort the function updates to the framework will be very limited

and will allow for a slowly but steadily decreasing failure rate. The ideal situation is a non-periodic bathtub curve as baseline representing the effort invested into the framework which becomes stable over time. This is sketched out in the bottom diagram in Figure 80. Additional effort almost exclusively results from inserting research-grade prototype modules which obtain good stability through correct interfacing with the system. The quality of these modules is not relevant for the quality of the system as a whole. Early publications will be more expensive as the framework will still be under heavy development. However, when the system reaches a certain degree of stability and usefulness, publications will require significantly decreased development effort.

Stable system architecture obviously requires some effort up front. This is why SE is not very popular: it means shifting a lot of effort to very early stages of the development process, and the ensuing positive effect takes a relatively long time to manifest. What we need is just the *right dose* of SE. The framework needs to be cleanly designed and developed. Many other parts, however, like well-known basic algorithms required for some ground truth and baseline, can be implemented and integrated by anyone with a decent understanding. This can even be accomplished by involving students or interns. To control the influences of the different parts of the system, they simply need to be known first. As trivial as this sounds, this first step, installing a rudimentary configuration management, is often reduced to the simple use of version control systems for source code, e.g. Subversion[10]. As useful, even important, such a version control system is in general, for a software system's development of the complexity discussed here, more elaborate configuration management functions are required, e.g. test configuration definition, automatic testing, and approval or rejection of new code, which cannot be delivered by these tools.

4.2 VIS*lib* – a Reusable Source-Code Library

Simply reusing existing source code in new programs is common practice, even in working on proof-of-concept prototypes. The reuse of source code is encouraged, which in turn reduces development time and enhances the quality of programs [Mey94]. This is a first step, but it alone does not make it a minimalised SE, as the source code is not subject to a clean design or development, but evolves dynamically and chaotically on demand. But, this approach offers the opportunity to introduce just enough SE where needed to further optimize software quality. This concept was originally presented at the ReVisE workshop at the IEEE Vis Week 2009 [GMRE09].

Class libraries are the most common way of source code reuse and are widely accepted to be beneficial to reducing development time and increasing software

[10] http://subversion.apache.org/ (last visited 30.12.2011)

quality. While public class libraries like *boost* aim at maximizing generality, which often comes at the cost of slightly suboptimal performance, for many software applications, like in interactive visualization, runtime performance is of utmost importance and speed-optimized implementations of classes that could also be found in common libraries are required. These implementations are not generic enough to be widely used, but they are often valuable assets for the academic group they originate from. As there usually is a strong coherence in the working areas of the members of the group, these implementations are important as basis and ground truth for on-going research work. Thus, such algorithms must not be lost in a single prototype program, but should be collected in coherent libraries. To successfully collect and reuse valuable source code within a research group, several conditions must be met:

First, when working on a new visualization application, even if it is just a small prototype, the developer must be aware of the possibility of reusing code and creating reusable code from the start (i.e. following the concepts described below throughout the whole development time; not just *cleaning-up later*).

Second, parts that have been identified for possible reuse must be collected in the research group's repository and therefore require some additional work: it is crucial that these implementations are encapsulated in a cleanly defined interface following the concept of information hiding [Par72] (which is inherently supported by modern programming languages) and optimally stamp coupling [PJ88] only. Such design requires some thoughts put into them and the resulting implementation is usually slightly larger than just for the present application.

As *third* condition, all classes collected for reuse must follow some coding standards and, most importantly, must be reasonably documented. As it is the very nature of research groups to have high fluctuation of personnel, it is essential that the source code is readable, maintainable, self-contained and works out-of-the-box, to guarantee efficient reuse and to make the work reproducible [Wil06]. For the same reason, a complete API documentation, that particularly highlights assumptions made for performance reasons, is indispensable. Tools like *doxygen*[11] are reasonable utilities in this context.

The *fourth* condition is to be aware of platform independency. Optimized implementations often require direct access to operating system functions, making even migrating between two Linux distributions non-trivial. Hiding different implementations behind interfaces greatly alleviates porting applications, which is a major problem once platform-dependent, quickly developed applications have reached some size. Implementing the same functionality for all platforms at once helps to ensure the same behaviour on all operating systems.

[11] www.doxygen.org/ (last visited 30.12.2011)

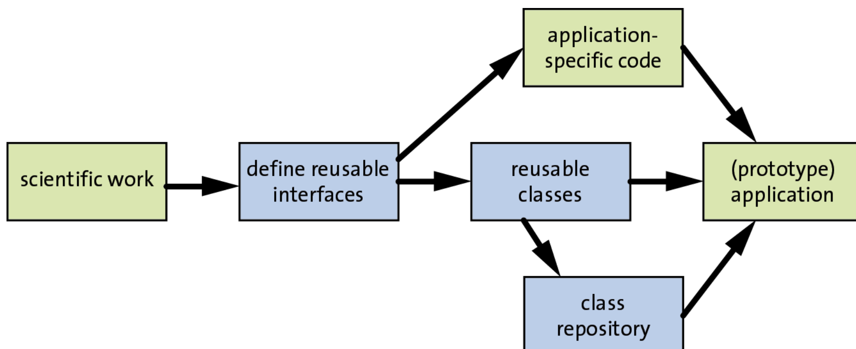

Figure 81: Schematic illustration of the proposed development process: Instead of monolithic prototype implementation reusable classes are identified and integrated into a local repository. Only for these classes additional effort is required guaranteeing high source code quality, like thorough documentation.

If these conditions are met, or at least if the developers are aware of these issues, a light SE process to only engineer classes to be reused when necessary can be installed. The process is illustrated in Figure 81. Increased development efforts (green boxes) are only required for identifying reusable parts and when writing the reusable code. This effort decreases in time as the number of classes to be added to the repository also decreases. Establishing such a process across (competing) institutes would require extensive management effort and will incur problematic aspects. As much as this would be beneficial for all, it is not required. Instead each research group can set up such an internal repository.

This class repository must be organised sensibly to aid the whole process. Otherwise it will not be more than a mere storage of source code text files. If structured properly, a central storage point and documented classes also implement a self-organizing knowledge management supporting the on-going development. To do so, the source repository should be divided into multiple libraries with clean dependencies. A base library is required to mostly deal with fundamental data structures, which can be used for data exchange through the interfaces of further reusable modules. At least facade definitions are required here. The remaining classes can be designed in tiers above this core, which are all interoperable, but become more and more specialized for the specific task. Thus, the whole library is organized into modules of high cohesion which can be used independently.

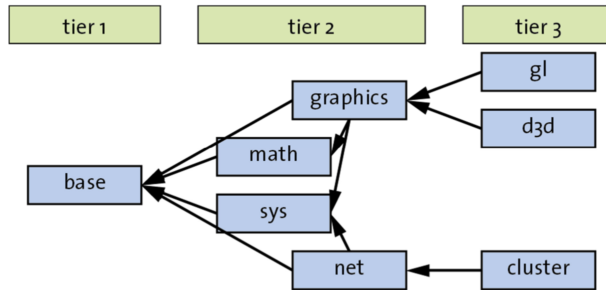

Figure 82: Module structure of the VIS*lib*. The modules, which are implemented as static libraries, are organized in three tiers building-up on each other.

Within the VIS research group such a library, namely the VIS*lib*, has been installed and has proven to be practical in everyday work. The library was started by collecting valuable code pieces from graduate students of the group. After the interfaces have been made coherent and comprehensively documented, these classes form the inner core of the library, called *base*. Some example classes of this module are *String*, *Exception*, *Array*, *RawStorage*, *Pair*, *SmartPtr*, and others. These classes serve as base classes for derived implementations as well as basic data types. For example, all exceptions thrown from implementations within the library are derived from *Exception*, which provides basic operations to get the exception's message and origin. *RawStorage* is a basic type capsuling an unformatted memory block. It, however, provides convenience methods for byte-addressable but typed access to sub-portions of the block, as well as content-preserving size changes. Although all of these classes are, in some form, available in other libraries, these base definitions are still required for consistent and interoperable interfaces throughout the library. The structure of the whole library can be seen in Figure 82. The individual modules of the VIS*lib* are organized in three tiers, which get, from left to right, more and more specific. Tier 1 only contains *base*, as this module defines all required types and utility functions.

Building on this *base* module, four further modules are constructed in tier 2: *sys*, providing platform-independent interfaces to operating-system-specific implementations, like *Threads*, their synchronization mechanisms, e.g. *CriticalSection* or *Events*, *Files*, including optimized memory mapped data transfer (*MemmappedFile*), or *PerformanceCounter* to access a high-precision clock. All these classes are utility classes only wrapping the corresponding calls to the operating system functions. The main problem is providing the same functionality on all supported platforms, in this case, Windows and several Linux distributions. Sometimes, only the commonly available functions are implemented and some functions will throw *NotImplementedException* or *NotSupportedException* on some operating systems (which is documented). Some classes, like *RegistryKey*, are only available on

some operating systems (Windows only in this case). Closely related is the *net* module, which contains corresponding classes for low-level network communication, e.g. a *Socket* interface definition compatible with IPv4, IPv6, and InfiniBand. Utility classes to access *DNS*, implementing a simple *TcpServer* or a *SimpleMessage* increase the usefulness of this module.

Another second-tier module is the *math* library, which contains math-related classes in the context of visualization and computer graphics. These range from simple classes like *Vector*, *Point*, *Quaternion*, *Matrix*, *Line*, *Cuboid*, etc. to more sophisticated and specialized classes like *FastMap* (dimensionality reduction), *ForceDirected* (graph layout) or *pcautil* (collection of utility functions for principal components analysis). All classes are implemented as templates, allowing for different type instantiations (i.e. most of the time *float* or *double*, sometimes also integral types) as well as for different memory layout and storage: e.g. a *Matrix* can be stored row-major or column-major to make its internal representation match requirements of API functions of other libraries. All basic types, e.g. *Vector* or *Point*, can be assigned raw memory pointers, which will then be interpreted accordingly.

The *graphics* module is the last second-tier module. This is a collection of utility classes and interfaces for generic drawing operations. The two third-tier modules building on graphics are *gl* and *d3d*, which specialise the provided classes either for OpenGL or for DirectX. Apart from type classes, like *ColourRGBAu8*, and utilities, like *FpsCounter* or *Cursor2DRectLasso*, two of the probably most important classes are *Camera* and *CameraParameters*. The latter one stores parameters of a camera in 3D space, like position, direction, and field-of-view, including advanced parameters for stereo and Powerwall support, like image-space tiling or stereo disparity. The *Camera* class acts as facade object to this parameter object and provides operations to evaluate these values. E.g. providing a scene bounding box, the camera can automatically adjust its near and far clipping planes accordingly. The two derived camera classes in the *gl* and *d3d* modules are able to compute matrices from the camera settings in the corresponding memory layouts. *CameraParameters* allows for easy serialization of its content, provides update events, and can be connected to classes derived from *AbstractCameraController* to be changed in a consistent way. On example of such a controller is the *CameraRotate2D* which translates movements of a normal mouse cursor to rotations of the camera. Instead of the usual monolithic camera interaction codes usually found in research prototypes, a programmer can compose many interaction patterns by simply connecting the corresponding classes, and essentially only putting mouse coordinates into the system.

The last module of the library which will be discussed here is *cluster*, a tier three module building upon *net* and containing special utility functions to support distributed rendering on CPU compute clusters. The most important classes here implement a simple message passing across cluster nodes, implement rudimentary parallel rendering for OpenGL and Direct3D, and provide an udp-based *DiscoveryService* to collect cluster nodes on a software service-based search.

Started in 2006 with some base classes, primarily developed by Christoph Müller and myself in parallel to our normal work, the library has grown to roughly 1300 files, 2800 classes and 250,000 lines of code at the end of 2011. Most of the collected functionality was already available in other libraries. The VIS*lib* however aimed at consistent interfaces among all functions, and high performance favoured over robustness against misuse. Third party libraries are only used sparsely, but used where possible as to not *reinvent the wheel*. With the comprehensive functionality this library has proven itself to be highly valuable for everyday implementation work and is one of the main pillars MegaMol builds on.

4.3 MegaMol – Framework Design

For interactive particle-based visualization of large MD data sets, none of the existing tools had the functionality, flexibility, or design to be used as basis for the research work planned in the SFB 716 [DFG]. Thus, as already mentioned in Chapter 4.1, based on previous work on research prototypes and heavy usage of the VIS*lib* the MegaMol project was started with the goal to construct a flexible visualization utility, useful to cooperation partners in the SFB, and forming a development platform for the research on visualization of large MD data sets. Given this principal requirement and the fact that more specific requirements will not be available, which is common in software industry and especially intrinsic to research work, the need for a fitting design emerges. In addition, the process of implementation, re-use of existing code, re-write of existing code, and prioritising the development of missing components, seemed overwhelming if not planned properly. Thus, this chapter sketches such a development process for *visualization systems engineering*. It is a combination of structured design [CY79] and object-oriented design [Boo04] adapted to the practical needs in visualization research. The SE formalisms are explicitly reduced to obtain a reasonable trade-off between software quality and development time.

The overall structure of the development process is outlined in Figure 83. The three main phases are: analysis, design, and implementation. The most important aspect of a system is the framework design, as a good design will support many functional modules before the need for improvements and major extensions arises, which finally may lead to a re-design and thus iteration of the whole development process. But even when rewriting the framework most of the individual functional modules can be reused without any changes, if the first interface definitions were reasonable and were strictly following the concepts of data hiding and minimal coupling (e.g. stamp coupling). Of course, no explicit, direct answer or concrete development plan for all system developments can be given. Instead, the concepts presented here form a guideline to point out which questions and which choices to consider in order to obtain a sustainable system architecture.

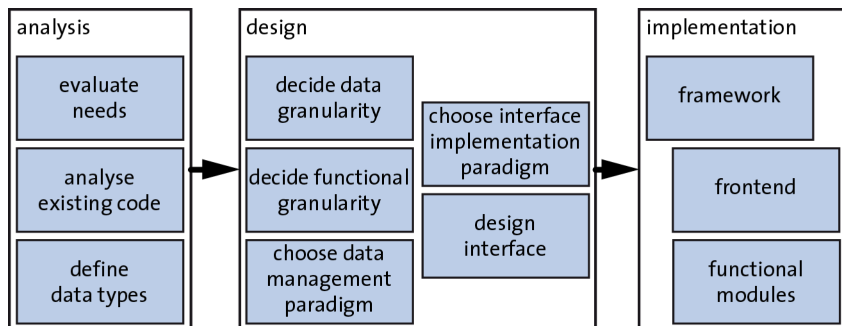

Figure 83: The three phases of the process for *visualization systems engineering*. Subtask sequence is shown by the horizontal position; vertical ordering is arbitrary and denotes possible parallelism.

In the first phase, the *analysis* phase, the SE formalisms normally employed can be simplified and the required activities can be carried out in almost arbitrary order. The whole process within this phase is driven by some kind of goal, which needs to be specified either by specific research application scenarios or at least a class of visualization algorithms that need to be supported by the resulting system. This goal serves as guide for the design decisions. Additional requirements stem from the possible integration of existing code, e.g. available research prototypes. The system must support suitable data flow and control flow mechanisms to make this integration possible. The definition of the required data types, e.g. grids, particles, or graphs, concludes the analysis phase.

The *design* phase contains the probably most important work, as here the flexibility, extensibility, as well as the limitations of the final system are determined. Since visualization software is mainly data-driven, e.g. the visualization pipeline itself describes a stage-wise data flow, a software design method should be chosen accordingly. Two fitting design strategies are *object-oriented design* [Boo04] and *structured design* [CY79]. Object-oriented design fits a data-centred view very well, as the data is an object which is transformed, parameterised, and passed on. In addition, structured design complements this approach by strongly focusing on the data flow, i.e. which modules communicate, which data is passed on different occasions, and how concise interfaces can be defined. Detailing both methods is not possible within the scope of this thesis. Instead the important aspects concerning visualization systems will be discussed.

Following object-oriented design, the data objects of a visualization system define the *granularity of objects*, e.g. data set as a whole, intermediate grouping, like molecule clusters or spatial grid cells, individual atoms, or their attributes or interactions. This choice influences on one hand the performance of the visualizations

and on the other hand the addressability of the data entities. For a concrete application, these two aspects must be traded off: if each individual atom can be acted upon, it is possible to perform fine-grain filtering, mapping, and analysis operations. However, this will also result in significant storage overhead as well as high processing and rendering times compared to chunk-wise processed data, which most likely will prohibit interactive rendering of large data sets. Chapter 2.2.1 showed that block transfer is always beneficial for particle data sets, e.g. from MD simulations (cf. immediate mode rendering). If the particles are not stored in a linear memory layout, the best way to transfer this data to the GPU is mapping VBO memory and copying the data required for rendering, which is, however, inferior to VA data transfer. In contrast, storing the whole data set as one linear memory block allows for fully utilizing VA transfer capabilities, but hinders advanced operations like the block-wise culling of the data (cf. chapter 2.3). Therefore, the best suitable granularity must be chosen depending on the expected data file sizes, keeping in mind that the data files are continuously growing, and the required addressing capabilities for filtering, mapping, and analysis. As MegaMol focusses on large MD data sets, the chosen data addressability is rather coarse: in general a single time frame of a trajectory is used as a whole. Individual atoms are either referenced by their index value or by their implicit, relative in-memory position. The data set might be further subdivided using a spatial grid or hierarchy (cf. Chapter 2.3, or for the use of quantization in Chapter 2.2.1). With any such structure, the time frames, already completely stored in main memory, must be fetched as a whole into a processing or rendering module, but then only a corresponding subset can be selected, e.g. for transfer to the GPU.

Data modification operations are not only related to the data granularity, but also to the chosen data management paradigm. The right paradigm even allows further adjusting the granularity, but considering the focus on large data sets, the paramount goal is to avoid duplication, while still enabling data manipulation, e.g. adding further attributes, while still storing the data in an optimized memory layout. The two probably most distinctive data management paradigms can be named: *volatile entity handover* and *parameter-enriched data views*. *Volatile entity handover* strictly follows the producer-consumer concept and enables true pipelining. Input data can be modified, filtered, or enriched arbitrarily. Memory ownership is passed along with the data itself. Data is only duplicated when required: i.e. the visualization pipeline is extended to a network, e.g. for coordinated views. In *parameter-enriched data views*, the original data is never changed. Instead, additional properties are attached to views (cf. the corresponding concept in data bases) using synchronized arrays (i.e. arrays of the same length in which the entries at the same indices are interpreted as attributes of a single data entity). Elements within the data are thus referenced by index lists. On the one hand, this means there will never be any duplication, but on the other hand, even reduction operations like subsampling, filtering, or abstraction will increase the amount of data (local to the view).

The access to the data might also be more complex, due to the de-localization of all required information into several arrays.

Both variants require a mechanism for caching of intermediate data to avoid computation overhead. Using volatile entity handover only the final data, right before presentation, needs to be cached, if the whole calculations are unlikely to be repeated for the data. This approach is most often used in visualization tools which generate classical mesh representations of the data, e.g. [AVS]. However, if the data changes often, like when visualising data in-situ from a simulation, the whole pipeline must be re-run continuously. For parameter-enriched views, the partial data produced by each module needs to be cached. This approach is beneficial when exploring large but static data sets, as e.g. adjusting visualization parameters like iso values for isosurfaces will only change the output of this single module (and dependent modules) and do not require updating the original data. MegaMol follows the concept of parameter-enriched views. Data is never accessed directly, but through facade objects. Adding further attributes to the data results in only replacing the facade object and attaching different memory segments to it. The memory ownership remains always with the originally producing module, e.g. the data file loader.

Closely related to the data granularity and management paradigm is the selection of the *granularity of the functions*. The smallest sensible scope of a function is derived from the data granularity: functions always working on small fractions of the passed data will likely waste performance or space. Contrariwise, if only extremely large multiples of the data entities can be processed but only the results of a specific subset are relevant, the data transfer or memory requirements are increased needlessly, e.g. significant transfer overhead to the GPU or expensive filtering on the CPU incurring data duplication. Thus, functions should represent a sensible semantic unit as postulated by structured design, and should fit the data granularity. Function granularity and data granularity are thus interdependent and must be selected together to the meet the needs of the software system to be created.

A basic concept of function design is to handle all functions as *black boxes*: a user or developer using the function should never need to consider the internal implementation of the function itself, except for behaviour specified by the function's documentation. Additionally a function's parameters should also match the required granularity, making the function's interface as a whole fit for the function's purpose and making functions with same behaviour, but potentially different implementations interchangeable. This concept can, of course, be adjusted to aid high performance implementations. For example, consider the extraction of ellipsoids for molecule clusters in chapter 3.1.1. While a function could compute the parameters for one ellipsoid from the positions and radii of the particles of one cluster, for performance reasons the implementation in MegaMol takes the whole data of the time frame and computes all ellipsoids for all molecules clusters at once. This allows for utilization of parallel multi-core CPUs and better fits the chosen data granularity in MegaMol.

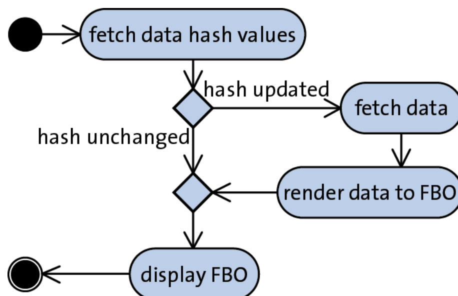

Figure 84: Sequence diagram of a simple rendering module; the resulting image is cached in a FBO and only re-rendered if the data changes, which is detected by testing the data hash value provided by the source module.

The *interface implementation paradigm*, as the name suggests, defines the concrete implementation concepts for the interfaces used within the system. Basically, the control flow needs to be chosen as well as the way of function composition. The two basic control flow possibilities are the *push* and the *pull* paradigm. The *push* paradigm, often employed by visualization systems, propagates data from the producer to the consumer as the data becomes available. Often combined with volatile entity hand-over, caching is required to ensure interactivity of the presentation by shortening the pipeline, seen from the output end, as much as possible. The *pull* paradigm, favoured by computer graphics applications, is steered from the output end. Data is requested on image refreshes, often periodically triggered for animations. Caching is required to avoid continuous re-computation and mask data starvation. Which of both paradigms better suits a system is mainly decided based on which stages of a corresponding computation pipeline change more often: If the early stages, like data acquisition, e.g. data set loading or data update from simulation, tend to change more often, the push paradigm is beneficial. If the late stages, like mapping and rendering are more likely to change, the pull paradigm is advantageous. In both cases mechanisms for propagation of update events, i.e. *new image update required* vs. *new data is available*, will be required. Such a mechanism makes both approaches work similarly. MegaMol was implemented following the pull paradigm as it focuses on the representation of particle data sets and many functions were inspired by methods from the field of computer graphics. The output window is refreshed periodically for animations, changes in camera parameters, etc. Different caching stages are implemented; e.g. derived representations are only recomputed if the data from depending modules changes. Data updates are detected by comparing data hash values provided by the producing modules. The concept of pull control flow and update checks is sketched for one rendering module in Figure

84. Certain elements, like user-changeable parameter values, also trigger update events, which call corresponding functions of the related modules.

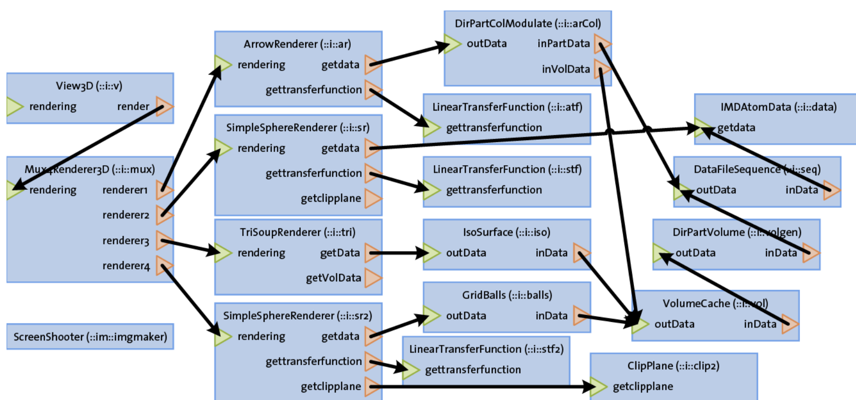

Figure 85: Example of a MegaMol module graph. The entry module *View3D* is connected to a *Mux4Renderer3D* which combines the results of four different rendering modules. Utimately, data is only loaded from and then provided by the Module *IMDAtomData* loading a particle data file of the corresponding format. Other modules enrich the corresponding view of the data (e.g. *DirPartColModulate* add a new colour channel) or derive new data (e.g. *IsoSurface*).

The composition of functions for the visualization application can be either carried out at compile-time or at run-time. A dynamically composed system that performs its connection checks of the interfaces at run-time generates some overhead, but is the most flexible variant. It can be distributed in binary form and the addition of modules can occur afterwards and by application needs. In comparison, checks performed at compile-time improve the efficiency of a system, but this approach results in a static selection of functional modules. Each new application scenario requires reconfiguration of the code and compilation, but implementation is much simpler and often less error-prone. The question which method to use is not very important. A first instance of a system can be compile-time composed. As long as the module interfaces and the data management are defined cleanly, the system's framework can later be refactored to support run-time composition without requiring significant changes inside the modules' implementations. For MegaMol a run-time composition of module was implemented, even including a plug-in mechanism, which allows loading of different functional modules from different binary libraries (DLLs on Windows or SOs on Linux). Figure 85 shows a module graph set up in MegaMol. On the left the *View3D* represents the output window including a 3D camera setup. The four modules in the middle (left) are four renderers which produce graphical elements in a frame buffer managed by the *Mux4Renderer3D* mod-

ule. All data used by all representations is originated from the particle data loaded by the module *IMDAtomData* on the far right. The arrows show the interface connections and thus the control flow within this module graph. The directions on the arrows show that MegaMol follows the pull paradigm, requesting updates from the depending modules.

The final design decision to be made is the *definition of* the individual functions' *interfaces*, depending on the granularities of the data and the functions. These definitions must be made following structured design and must avoid content coupling and global coupling. The concrete implementation mechanism for the interface connections, e.g. signals and slots, callbacks, or functors, can be chosen arbitrarily as it is mostly a matter of taste. The interface definitions must be based on the semantics of a function and only secondarily on the format of the data. Although optimal data transfer is of high importance, independent interfaces will allow for changes in the framework without changing the function implementations, thus enabling for framework rewrites if required. As compromise additional interface extensions can be introduced to provide convenience methods to access the modules functions. However, the interfaces should be as small and as simple as possible to aid code maintenance. Any function which is not required but *could be useful in the future* and which would be rather simple to be added later on should be omitted at first. It is almost impossible to subsequently remove a function from an interface without replacing the interface as a whole.

In MegaMol, interfaces are implemented by *Call* classes. Interface connections are formed by *CallerSlots* and *CalleeSlots*, objects similar to the signal-slot mechanism of Qt[12]. Type safety is implemented by identifier string comparison through additional run-time type information functions. The connection of two slots is formed by an object instance of the corresponding *Call* class. A *CallerSlot* is connected to exactly one *Call* object. Through the slot only this *Call* object is visible and not the connected module object. A function is called on the *Call* object which in turns calls a corresponding callback on the connected module. Multiple *Calls* may be connected to any *CalleeSlot* as the currently calling *Call* is provided as parameter for the corresponding callback function of the called module. These callbacks usually perform a calculation, or fill the *Call* object with meta data and data pointers to data owned by the module, following the concept of *stamp coupling*. The calling module can then access these pointers, usually in a read-only fashion, through the *Call* object. Thus, the Call classes define the function interfaces and the function granularity, as well as the passable data structures and thus the data granularity. The modules implementing these interfaces, by providing corresponding *CalleeSlots*, are arbitrarily exchangeable. One example interface is the *CallRender3D*, which is called with a reference to a render target, either FBOs or frame buffers, to make a renderer module produce an output image. Additional data provided are viewport sizes and timing values for animations. A second example, which is more data

[12] http://qt.nokia.com/ - Cross-platform application and UI framework

related, is *MultiParticleDataCall*. This interface transports a list of linear memory segments which hold particle data and meta data on how to interpret the particles, e.g. using float or double coordinates, or the object-space bounding box of the data. The *SimpleSphereRenderer* uses this interface to request data. Any data file loader or any data generator can be used as source. MegaMol is thus an optimal platform for visualization research implementations, as the flexible high-level interfaces allow for optimized module implementations, exchangeable modules and extendable call graphs.

The last and probably largest phase of the system development process is the implementation. Here, the framework, a frontend to the framework, and the functions themselves need to be implemented and integrated. Although the implementation order is more or less arbitrary, the framework needs a bit of lead time, as the frontend and especially the functions rely on it. After that the modules' functionalities can be added. Frontend modules form the application entry points of a system and steer the data flow in the system. Therefore they are a vital part of the framework and must be engineered equally carefully. The other modules are either recycled from proof-of-concept prototypes by adding interfaces that fit the system, or implemented from scratch. The MegaMol framework was re-designed and re-implemented four times. Each time more flexibility was added, e.g. re-design 0.3 added support for run-time plug-in libraries, however, the function interface *Calls* were never changed significantly. In contrast, modules and calls were added, removed and changed continuously.

MegaMol separates the module graph used for visualization from the frontend into two binary modules. The framework, and thus the main functionality of MegaMol, is implemented in a *core* library, which is the central dependency source for plug-in libraries. This core implements the advanced run-time type information system, the run-time module graph management as well as most of the basic calls and modules. The core itself is not an executable, but a DLL (Windows), SO (Linux). The executable is the frontend, which loads and uses the core library. The frontend is responsible for generating the rendering contexts and to publish the adjustable parameters to the user. For doing so, the core provides several utility functions. This allows for different frontends for different application scenarios, like the *console* frontend meant for development and Powerwall usage, which thus has only limited overhead and simplifies debugging of the core or of plug-in libraries. Other frontends focus on specific applications, e.g. in the context of protein visualization. The lower-left image in Figure 86 shows such an application frontend to analyse cavity evolution, especially formations of openings, in time-dependent protein MD data sets. These frontends were partially implemented by students. These different frontends allow using MegaMol in the context of visualization research as well as in easy-to-use applications for end users.

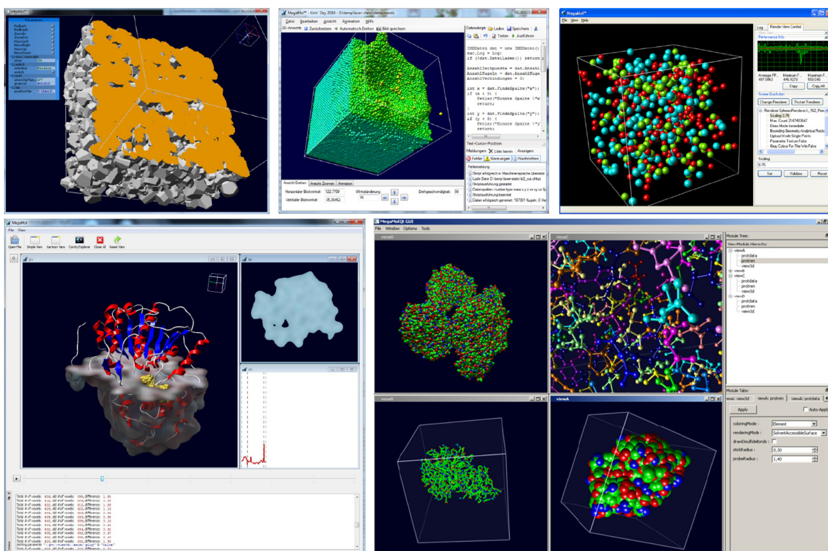

Figure 86: Different GUI frontends of MegaMol: *top-left*: the current MegaMol Console frontend for debugging and Powerwall support; *top-centre*: a specialized application for teaching, in which the mapping operations scripts can be written in C#; *top-right*: an older GUI of MegaMol (V.:0.2), written with Windows Forms; bottom-left: a GUI written in Qt for the application of cavity analysis [KFR+11]; *bottom-right*: A GUI supporting multiple coordinated views, also written with Qt.

5 Conclusion

The methods presented in this thesis focus on the interactive visualization of large, time-dependent particle data sets on commodity desktop computers. Such data sets mainly result from MD simulations, which are a principal tool in many research areas. The simulation setup sizes and thus the sizes of the resulting data sets are quickly increasing, because of the increasing availability of cost-efficient compute hardware and advances in the simulation algorithms. Interactively exploring such data sets in an everyday working environment poses a significant challenge in terms of pure rendering performance and, even more importantly, in terms of effective and efficient visual representations.

Point-based GPU glyph ray casting has been established as state-of-the-art method for creating such visualizations. The fundamentals are shown in chapter 2.1 and all stages of the applied rendering methods have been thoroughly optimised. Chapter 2.2 shows extensive analysis results on the communications limits between secondary storage, main memory, and the GPU. A two-stage culling technique, presented in chapter 2.3, allows working with data sets of up to 100 million particles. However, the disadvantageous glyph-to-pixel ratio results in poor visual quality due to aliasing effects. An image-space normal vector estimation incorporated in a deferred shading pass and object-space ambient occlusion, both presented in chapter 2.4, allow emphasising the structures implicitly formed by the particle data. Extending the rendering from simple quadrics to glyphs composed from basic quadric shapes was discussed in chapter 2.5.1, together with the optimisations required to render large numbers of such glyphs interactively. Polyhedral glyphs were discussed in chapter 2.5.2.

However, whenever possible, high-level structures defined based on the original particle data should be extracted to create concise visualizations which are more efficient in providing an understanding of the data. The example of molecule clusters in the context of thermodynamics is discussed in multiple variants. Chapter 3.1.1 presents ellipsoid glyphs, computed by principal component analysis of the particle data segmentation results of different cluster detection criteria. A better surface representation can be achieved by metaballs, for which two implementations are shown in chapter 3.1.2. A corresponding approach for isosurface ray casting with partial on-demand volume reconstruction is presented in chapter 3.1.3. Chapter 3.1.4 shows one possibility to extract and ray cast a molecule surface constructed from distinct geometrical elements. All computations are completely carried out by the GPU. As further example structure extraction and visualization of dislocations, irregularities in atom lattices of solid metal MD simulations, are presented in chapter 3.1.5. The network topology formed by dislocations and stacking faults, identified though local neighbourhood analysis, is extracted by a segmentation method, based on simplification of a neighbourhood graph. From the segmentation result spline tubes and semi-transparent flat planes are extracted, which provide a clean and easily perceptible visualization.

As extension of these spatial extractions, several approaches are presented to visualize the dynamics of particle data sets in static images. The LCCVD algorithm, presented in chapter 3.2.1, allows reducing the number of particles by several orders of magnitude while preserving the particle density distributions. The generated reduced particle sets can be stably tracked through the trajectory and allow to extract representative path lines capturing the behaviour of the original data. A specialised approach was developed for path lines of solvent molecules near proteins, e.g. in cavities formed by the proteins' atoms. The principal path lines network of the solvent molecules' trajectories can be extracted using an atom-speed-based clustering, which is detailed in chapter 3.2.2. When abstracting from the physical space more generic solutions can be defined. For example, molecule clusters can be represented in diagrams of their evolution over time, including their interactions, i.e. direct exchange of molecules, which is shown in chapter 3.2.3. The presented definition of flow groups, particles which are exchanged between two molecule clusters during one period of time, allows for characterizing the interactions of molecule clusters and especially allows for comparison of the results and stability of different cluster detection criteria.

All visualizations presented in this thesis were developed in close collaboration with researchers from the corresponding application domains. To nurture the dialog and to aid the process of understanding between the collaborators it is most helpful if both parties can use the visualization tool on their own. However, most scientific visualization tools written in academic environments are of poor software quality. They often lack usability and thorough error handling making them tiresome for non-computer scientists to use. Chapter 4 discusses how software quality of academic programs can be increased to be fit for the future challenges of even larger and more complex data sets and scenarios, which both require complex software solutions, e.g. utilizing multi-core processors or heterogeneous compute environments. MegaMol, a visualization platform for large, time-dependent particle data sets, was developed in the SFB 716 following these principles, resulting in a high quality software system. To develop such complex software the concepts of software engineering (SE) are needed to manage the work. In chapter 4.1 the principal problems of SE in research environments are discussed, and chapter 4.2 presents a simple concept on how to decide when to apply SE processes. The simple concept of a repository of re-usable source code is formalized. The VIS*lib*, a class library created at the Visualization Institute of the University of Stuttgart during the development of MegaMol, serves as concrete example. For large software, such as MegaMol, the most crucial aspect is its architecture. Chapter 4.3 presents some basic guidelines especially tailored for designing visualization systems addressing this important aspect. These rules are the result from both the knowledge of SE and the experience of designing and maintaining the MegaMol system framework.

Several visualization applications were implemented using MegaMol. Some of them are promising for future work and future publications, but have not reached a suitable state to be presented in this thesis. One visualization example is

the extraction of a cavity networks within porous media constructed from the polyhedral glyphs presented in chapter 2.5.2. The void space needs to be collected, but has no representing elements within the data. As resampling into a volume representation should be avoided, due to the need of very high resolution and adaptive sampling, an approach following spatial subdivision with a binary space partitioning tree is currently under development. This tree can represent the data as a whole including the void space. The result will be a topological network, similar to the dislocation networks shown in chapter 3.1.5. This will allow for analysing parameters like connectivity or pore volume independent from any sampling resolution. A further example stems from the continuation of the collaboration concerning oriented particles (cf. Figure 2 and the arrow glyphs in chapter 2.5.1). The overall structure of the data set must be emphasised without losing the information about the individual particles. The approach being currently evaluated is based on a scalar field computed from the orientations within local neighbourhoods. This additional data can be used to enrich the glyph representation by additional mapping and by further graphical primitives, like isosurfaces. Many more examples could be listed here, especially from the field of protein visualization in the context of biochemistry. Michael Krone working in the SFB 716 in project D.4 on these visualizations uses MegaMol as basis for his implementations as well.

The further development of particle-based visualization will likely go into two directions: first, the direct rendering of particle data is needed to be enhanced further to be able to visualise the data sets, still increasing in size, at interactive frame rates. Second, advanced visual metaphors and abstractions are needed to create meaningful, efficient visualizations. The glyph-based visualization will always be needed, as it is true to the original data. Any derived visualization using abstract metaphors needs to relate to this direct representation as ground truth. Combining both types, direct and abstract visualization, and blending between both types of visualizations will also increase the acceptance of abstract visualizations among the application domain experts. Optimizing the glyph-based rendering is highly coupled with the advances in the computer graphics community. New functions of upcoming GPU generations may provide further improvements. Apart from this, the limits of a single GPU are reached with the methods presented in this thesis. To support even larger data sets, the parallelism scheme needs to be extended from within one GPU to multiple GPUs in a single machine, and GPU cluster computers. As more communication stages will be introduced, the latencies will increase. For interactive visualization this will be one of the most important aspects.

To create efficient visualizations, however, at some point the visual elements need to abstract from the original particle data. These abstractions need to be created in close collaboration with the application domain experts to meet their specific needs, usually focusing on the important attributes of the data and visual metaphors the domain scientists are familiar with. However, with the increasing complexity of the simulation methods, e.g. in multi-scale and multi-physics simulations, the number of relevant attributes also increases, making this reduction non-

trivial. This is especially true for visualizing the dynamics of time-dependent data sets. Combining concise representations, like crystal defect networks (cf. Chapter 3.1.5), together with the possibilities of multi-dimensional visualizations, like scatterplot matrices or parallel coordinates, might result in useful tools. However, the concrete approach to achieve this combination is not clear yet, as trivial coordinated views might not be able to present the different visualizations in a sufficiently related way. Especially the integration of time is not easy and might highly depend on the specific application task.

MegaMol, as a visualization research development platform for particle-based data sets, is perfectly suited to form the basis for such experiments with advanced, combined visualizations. However, MegaMol is not a suitable solution for every problem, as during development of this system some decisions were made which limit the capabilities of the software. Each system, existing ones as well as systems to be written in the future, has its advantages and disadvantages. Instead of using an existing system which almost fits the requirements everyone should seriously consider writing a new system which matches the requirements perfectly. This is not so much about *reinventing the wheel*, but inventing a better wheel.

Working on scientific visualization software, even beyond the scope of this thesis, and especially for interactive visualization for large data sets or complex, maybe heterogeneous data, the challenges to be faced are often related to the ones from the field of computer graphics. Therefore, it is important to utilize the knowledge available in this field to develop optimized rendering methods for the visualization problems. For high performance rendering implementations, it is important to fully harvest all available processing power and to optimize the rendering algorithms accordingly, i.e. parallel and interleaved computations. While this is a currently active field for the GPGPU approach, classical computer graphics methods, and even CPU implementations strongly benefit from parallelism as well. A graphics-related example, which is well known but only hardly ever implemented – due to its' complexity – is multi-threaded OpenGL rendering. Apart from the obvious uploading of data, like textures or VBOs, inside a second thread, it is also possible to perform several GPU-implemented computations in parallel, e.g. the generation of data structures like the HZB (cf. chapter 2.3). While this was not necessary for the presented method, as this computation is performed while waiting for the HWOQs to return, other implementations might highly benefit from such an approach. Other ideas and approaches available from the computer graphics community are well known and regularly applied, like transferring data in blocks as large as possible, using VBOs to optimize the transfer load, and minimizing the required changes of the OpenGL states. The same is true for less low-level problems, e.g. data structures for in-memory representation of the data. Here, spatial data structures, e.g. octrees, are basically the same in both research fields, visualization and computer graphics, and the continuous advances are beneficial for all applications.

However, the availability of that many methods possibly applicable to any project a visualization researcher might work on results in the question of which approach to actually use. Most of the time, the differences are insignificant and the decisions can be made based on personal taste, e.g. it often is not really important whether to use a kd-tree or an octree, whether to use marching cubes or marching tetrahedra, etc. But sometimes these decisions do matter. Often limitations are given by the data itself or by a further algorithm which is to be applied. The worst thing that can probably happen is that a decision was made, e.g. for a specific data structure, and during the on-going software development this data structure proves to be unsuitable. This problem can only be addressed by good planning ahead of the software to be written, but it can never be completely eliminated, especially not in the development in academic environments.

Chapter 4.1 discussed how to apply methods from SE to approach these problems for the development of visualization systems. In fact, the question whether or not to write a system in the first place is also part of SE. It is completely OK to not write a visualization system but to quickly put together a research prototype, if this approach reaches the goal, e.g. a handful of images for one publication. However, following the concepts of SE one must never forget the initial goal and the development quality of software. A research prototype is of inferior quality and must never be reused as basis for anything else but other research-prototype-grade software. If such a prototype is to be added to a visualization system, it must either be rewritten, following the coding standards of a visualization system, or it must be cleanly separated from the system, in the most extreme case even by sand-boxing the execution environment.

Visualization system development is not a generally applicable solution. There are several technical, organisational and even political problems beyond the ones discussed in chapter 4. Most of them can be reduced to the fact that a software system needs to be embraced and carried by a wide group of people, in terms of development as well as in terms of usage, while, at the same time, a good software system, especially at the start of its development, needs to be thoroughly designed by a single expert architect or a very small group of architects, to ensure high stability and high quality of the systems' framework. This expert-centred development process is required for, at least, the early design phase, as here decisions have to be made which are critical for the system as a whole. Errors in the framework can hardly be corrected as soon as the system reaches a considerable size. Such errors were made while developing MegaMol, mainly due to the lack of experience. Three re-designs of the framework of MegaMol were carried out. The first redesign was necessary because the original framework was not capable of handling update-events (cf. chapter 4.3). The second update changed the management layer controlling the module instantiation and the communication with the user interface. The third rewrite had the goal to implement a plug-in mechanism and introduced extensive meta-data structures for the functional modules. All three redesigns required changes in all functional modules already implemented. Fortunately, the

first shortcomings were detected early after the project had been started and thus only few modules existed at this point in time. Today, MegaMol consists of multiple plugins, developed by different people, and has several hundreds of classes implementing functional modules. Although the MegaMol framework still has several design flaws, e.g. neither support for OpenGL 3 or 4 can be integrated nor multi-threaded rendering or multi-GPU can be support. However, it is currently not economically possible to perform a further redesign at the moment. From this experience two important aspects can be deduced which must be considered in a software system frameworks' architecture:

First, the module architecture and especially a meta-data mechanism for extensive and arbitrarily extensible run-time type information need to be chosen. This also includes designing connection mechanisms for the functional modules and their interface definitions. All required information should be collected within the source code in form of meta-data information. This allows for type checking, run-time call graph composition, and something as ordinary as a generated online help. Especially when the system grows, these different elements will allow to automatically and thoroughly detect inconsistencies. Such run-time meta-data of connection interfaces also allows to decouple the problem of interface definitions from the initial framework design. The types and granularity of the data which is transferred over the connections between modules can be defined when the actual module implementations are written. Only the modules need to agree on the details of the interface, like the data being transported via a pointer to a raw memory block, a complex spatial data structure, or even a handle to GPU-side memory objects. The framework itself only needs to know an opaque interface type and connection mechanism.

The second aspect which is important for the system's framework architecture is the selection of technical functions and limitations. It does not make any sense trying to develop a system which can integrate *everything*, in terms of functionality for visualization applications. One good example is the choice of a graphics API, i.e. DirectX or OpenGL. The available functionality is comparable in both and most of the conceptual differences are a matter of taste. However, a system supporting both graphics APIs will obviously be very complex, and it is doubtful if this complexity is justified by the very limited gain. Assuming OpenGL was chosen to support Linux, the question continues on which version of OpenGL to support, as the newest version (3.x and 4.x) follow very different programming paradigms than classical OpenGL (version 2.x). For example, VAs are no longer supported and VBOs are – even on the newest generation of graphics cards – not able to reach comparable performance. The increasing amount of graphics memory allows to store more and more of the data set directly on the graphics board. This will enable prefetching algorithms, like described in chapter 2.2.2 for the loading from secondary storage, to be implemented for the GPU data transfer route as well. However, many laptops and netbooks, which might be attractive devices for collaborative and ubiquitous visualization scenarios, are powered by Intel graphics chips, which currently do not

support the newer OpenGL versions. A similar issue exists for multi-GPU and multi-threaded rendering. All correspondingly made decisions define the direction of the system, excluding other technologies and options. Revoking these decisions in a later development step is practically impossible.

While these two aspects hinder the development of a system due to technical issues, the main problem for a system to be successful is a non-technical one. All the additional effort invested to start system development will only amortise if the system is used for many follow-up applications, ideally by several visualization researchers, as mentioned above. Motivating the usage of the system is not easy, especially during the early development stages, when the system can hardly help its users and instead demands lots of additional work. Many initially interested researchers will say that they reach their goals faster if the just return to their prototype software, and they are right, to some extent. Thus, the system will lose development power and most of these people won't return to use the system, even if it matures into a stable and usable state, because they remember the system to require more work than it is worth. In later stages of the development the situation is not better. Let's assume the system has reached a useful state. Then the system will also have grown quite big. A variant of the same argument as before will be used against using the system: it is easier and faster to do prototype-development than to learn the system. Again, this is true for the moment, but extremely short-sighted. However, as the prototypes are sufficient for now, i.e. the publication the visualization researcher currently works on, it is nearly impossible to convince him or her to use the system anyway to benefit in the future. Basically, the same arguments used to argue for starting a system development on our own, instead of using an existing system (cf. chapter 4.1), can be used to argue against using such a system at all in favour of using prototypes.

Motivating visualization researchers to join the development team of a system is also hindered by the processes of SE itself. Most visualization researchers tend, like most other people too, to see SE as unnecessary over-formalism for a rather easy task. Working on a software project as large as a visualization system, this, however, is not true, as is shown in this thesis. Thus SE methods are needed to ensure the quality, especially the maintainability, of the software to be written. These include measures for quality assurance and coding discipline, like coding style guides. With arguments like *with this style guide I have trouble reading my own code*, or *the style guide does not matter*, or *you can use an automatic formatting tool to do this* people will tell that they do not care about the code quality. If then the style guide is somewhat enforced, the people will leave the project and return to their prototypes, optionally harvesting the project's code for their own. This problem must be solved politically, as it is a social issue and not a technical one. The project owner, usually the software architect, does not have any political power over the other developers in the project, unlike a head of a company. Thus, he or she has to convince the fellow programmers of the benefits the system will grant them, and, more importantly, of the necessity of the SE methods, as unpleasant they may be.

Additional motivation can come from the head of the research group, but this is only possible to a very limited extent, as the development of production-ready software is no job content for researchers like Ph.D. students. Research work can highly benefit from a successful visualization system. All participating researchers and developers gain advantages from the combined work in form of synergy effects. After the basic functions for the framework have been implemented by joint effort it is possible to concentrate all the implementation effort for a single publication into one or two relevant modules, while still gaining a visualization tool with advanced features. But it is hard to prove this claim to motivate fellow researchers.

A possibility to address this issue might actually be to increase the amount of SE in the development process. As SE is not only meant to increase software quality, what it was mainly used for within this thesis, but also to increase the software development quality, this might be a way to provide early benefits for potential fellow developers. Especially further quality assurances, like automated testing and profiling environments, could be sufficient visible advantages. But also further methods in planning and controlling the software development process, like task planning or a configuration management could be of help. The biggest issue will again be the usability. Most tools providing the mentioned techniques are rather complicated to use, even for people with an SE background. People without such a background will not use these tools unless they have to. Therefore, providing tools which are easy to use, maybe even specifically tailored for visualization system engineering, is an important task for the visualization community as well.

In general, computer technology and thus computer sciences, including visualization, are still evolving fast. The current focus on multi-core architectures, e.g. GPGPU, and highly parallel visualization algorithms for very large data sets will change as further changes of the broadly available computer hardware will take place, e.g. heterogeneous processing units. Concrete methods and algorithms presented in this thesis will be superseded eventually. However, the need for visualization will remain, especially in research environment where unexpected and not understood phenomena need to be explored interactively. Similarly the need for effective and efficient visualization development will persist. Regardless of any visual metaphors which will emerge, the direct, glyph-based representation for particle data sets will remain relevant, at least as ground truth. For the near future point-based visualization and GPU glyph ray casting will remain the state-of-the-art methods to do so.

Zusammenfassung – German Abstract

Die Analyse und im Besonderen die Exploration von großen Simulationsdatensätzen profitieren von Visualisierungen. Wenn es nicht möglich ist bekannte, charakteristische Kenngrößen zu berechnen, oder wenn die Wahrscheinlichkeit besteht, dass räumliche Faktoren eine wichtige Rolle spielen können, dann werden direkte visuelle Repräsentationen der Daten ein erster und wichtiger Schritt des Analyseprozesses. Dies gilt besonders für wissenschaftliche Simulationen, die neue Methoden nutzen oder neue Szenarien untersuchen, welche möglicherweise unbekannten Phänomene enthalten. Partikelbasierte Simulationsmethoden, wie die Molekulardynamik, Smooth-Particle-Hydrodynamik oder die Diskrete-Element-Methode, sind beliebte Beispiele, da sowohl ihre Simulationstechniken, als auch die durch sie untersuchten Szenarien Gegenstand aktueller Forschungen in den entsprechenden Gebieten sind, z. B. in der Physik, der Thermodynamik, Biochemie, den Material- und Ingenieurswissenschaften. Die Größen der Simulationsszenarien, und als Konsequenz die Größen der entstehenden Datensätze, sind in den letzten Jahren stetig gewachsen, um Lücke zwischen den durch reale Experimente und den durch Simulation erreichbaren Längenskalen zu schließen. Dies liegt nicht nur an der erhöhten Rechenleistung einzelner Maschinen oder der höheren Verfügbarkeit relativ billiger Rechencluster, welche vor Ort eingesetzt werden können, sondern auch an den signifikanten Verbesserungen der Simulationsalgorithmen.

Die meisten verfügbaren Visualisierungswerkzeuge sind unzureichend optimiert um mit Datensätzen aktueller Größen umgehen zu können, da sie oft den gesamten Datensatz in den Hauptspeicher laden müssen oder ausschließlich dreiecksbasierte Rendering-Methoden einsetzen. Für partikelbasierte Datensätze hat sich Ray-Casting als gängige Methode etabliert und große Datensätze werden üblicherweise durch Streaming- und Out-of-Core-Methoden behandelt. In dieser Doktorarbeit wurden die Performanz-Engpässe beim Datentransfer, sowohl vom Sekundärspeicher, als auch zwischen Hauptspeicher und Graphikhardware untersucht. Auch die Einflüsse unterschiedlicher Kompressionstechniken, basierend auf räumlichen Datenstrukturen und Koordinatenquantisierung, welche die Grundlagen für Streaming-Techniken bilden, wurden geprüft. Punktbasiertes Ray-Casting wurde für unterschiedliche graphische Elemente, wie Kugeln, Zylinder, zusammengesetzte Glyphen und polyedrische Kristallite erweitert. In dem die notwendigen Berechnungen durch fortgeschrittene Culling-Techniken minimiert wurden, wurde die Darstellungsgeschwindigkeit optimiert, dass die interaktive Visualisierung von Datensätzen mit Hunderten von Millionen Partikel möglich wird.

Ein zweiter und möglicherweise wichtigerer Aspekt der Visualisierung solch großer Partikeldatensätze bleibt allerdings dadurch unberührt: das Erzeugen ausdrucksstarker und nützlicher Visualisierungen. Selbst wenn ein System in der Lage ist mehrere Millionen Partikel darzustellen, leiden die entstehenden Bilder üblicherweise unter Problemen wie Aliasing und Visual Clutter, welche eine gute Wahrnehmung und das Verständnis der Strukturen der präsentierten Daten er-

schweren. In dieser Doktorarbeit werden zwei Ansätze für fortschrittliches Shading und Beleuchtungsberechnung präsentiert, um dieses Problem zu lösen. Das Problem des Aliasing wird durch ein Bildraumverfahren angegangen, welches die Normalen-Vektoren für implizit geformte Strukturen schätzt, während die Wahrnehmung der globalen Strukturen und der Tiefenkomplexität der Daten durch ein speziell angepasstes Ambient-Occlusion-Verfahren, welches globale Beleuchtung annähert, verbessert wird.

Für effektive und effiziente Visualisierungen müssen wichtige, abgeleitete Strukturen der Daten extrahiert werden. Die so entstehenden, kompakten Visualisierungen vermitteln einen besseren Überblick über in den Daten enthaltene Strukturen. Da hierdurch jedoch die dargestellten Informationen reduziert werden, muss sichergestellt werden, dass keine wichtigen Daten verloren gehen und dass keine verfälschenden, künstlichen Daten durch die Methoden eingeführt werden. Zwei Arten abstrakter Darstellungen werden an mehreren Beispielen in dieser Doktorarbeit beschrieben: die Extraktion räumlicher Strukturen und Darstellungen der Dynamik der Daten. Die Beispiele der ersten Art reichen von Molekülcluster, Tröpfchen im Kontext thermodynamischer Nukleationssimulationen, Versetzungen und Stapelfehler im Kontext der Materialwissenschaften bis zu allgemeinen Oberflächenrepräsentationen, ähnlich den Moleküloberflächen in der Biochemie. Die Dynamik von Daten wird an den Beispielen von Molekülcluster-Interaktionen, dem Clustering von Molekülpfaden bei Wasser-Protein-Interaktionen, und dem Verfolgen ausgestoßenen Materials in Laser-Ablations-Simulationen untersucht und präsentiert.

Um sicherzustellen, dass Datenverfälschung bei abgeleiteten Visualisierungen nicht auftreten, ist es wichtig, Experten aus dem entsprechenden Anwendungsgebiet in die Arbeit an der Visualisierung mit einzubeziehen, da nur sie in der Lage sind, die Korrektheit und Nützlichkeit einer Visualisierung im Verhältnis zu den Originaldaten zu bewerten. Solche Zusammenarbeiten sind dann am effektivsten wenn die Anwender selbst auch die Benutzer der Visualisierungssoftware sind und beispielsweise mit den Parametern der Darstellung experimentieren können. Hierfür ist es jedoch notwendig, dass die Visualisierungssoftware auch von Personen benutzbar ist, die keine Computer-Experten sind. Besonders Programme aus dem Forschungsumfeld sind oft lediglich fragile Prototypen, welche als Proof-of-Concept für einen Algorithmus dienen. Meist gibt es keine einfache Benutzungsschnittstelle oder durchdachte Fehlerbehandlung. Erprobte Methoden aus dem Gebiet der Softwaretechnik bieten hier jedoch Lösungen an, um qualitativ gute Software zu erzeugen, welche von Endbenutzern direkt bedient werden kann und welche den Entwicklungsaufwand nur gering vergrößern. Diese Doktorarbeit präsentiert ein Prozessmodell, zusammen mit Richtlinien, um gerade genügend Software Engineering in den Entwicklungsprozess wissenschaftlicher Visualisierungssoftware einfließen zu lassen. Als Beispiel für die Effektivität des Ansatzes dient MegaMol, ein auf punktbasierte Visualisierung von Partikeldatensätze spezialisiertes Visualisierungssystem.

Alle in dieser Doktorarbeit vorgestellten Methoden zusammengenommen bilden eine einheitliche Lösung für eine effektive und effiziente Visualisierung großer Datensätze mit mehreren Hundertmillionen Partikeln, indem die Probleme der Darstellungs- und der Ladegeschwindigkeit, der Wahrnehmung, der Feature-Extraktion und Nachverfolgung untersucht werden. Ein für das wissenschaftliche Umfeld passender Entwicklungsprozess für die notwendigen, komplexen Visualisierungsprogramme wurde ebenso vorgestellt. Obwohl die in dieser Dissertation präsentierten Methoden sich auf die punktbasierte Visualisierung von Datensätzen aus der Molekulardynamik konzentrieren, erlaubt ihre allgemeine Ausrichtung auch einen Einsatz in anderen Anwendungsszenarien.

List of Figures

Figure 1: Point-based surface rendering of a scattered data representation of the Stanford Bunny at coarse resolution (left) [GP07]; classical scatterplot (right, top) of the "blunt-fin" data set and continous scatterplot (right, bottom) of the same data [BW08] 10
Figure 2: Detail of a visualization of orientations of polarized atoms in cracked solid alumina (unpublished collaboration with B.1 and B.2 of SFB 716). While the polarizable atoms are orientated randomly in the bulk, the dipole momenta align at the surface of the crack. 11
Figure 3: A planar stacking fault surrounded by linear dislocations within solid crystalline metal, all formed by individual atoms [GDCE09]; the structures are classified and coloured based on the common-neighbour analysis [HA87], identifying FCC structures, BCC structures, and atoms with otherwise arranged 12 nearest neighbours. 11
Figure 4: Thermodynamics MD simulation of super-saturated vapor (CO_2 (left) and R-152a (right)) forming molecule clusters (coloured cyan in the right image) as predecessors of droplets during the process of nucleation; usually these pre-droplets would be spherical because of surface tension. Because of the extreme preasure and the PBC the molecule clusters in the left image instead connect to each other across different instances of the simulation area as a stable end state. 17
Figure 5: Comparison of 2D sections (2.25 mm • 2.25 mm) experimental data (µ-CT; left) with reconstructed model (right) of Fontainebleau sandstone. Grains are shown in grey, while space in-between is shown in black. The reconstructed model is stochastically very similar to the experimental data. The degree of stochastic similarity was measured and documented quantitatively using numerous geometric observables described in detail in [LBFH10]. 19
Figure 6: A small porous media sample modelled from 10 000 particles of 100 crystallite templates. Each particle is a scaled and rotated instance of one of these templates. 20
Figure 7: Example diagram for typical visualization of data from particle simulation with one spatial axis: a velocity profile (top) and density profile (bottom) for Poiseuille (solid lines) and Couette (dotted lines) flow of liquid methane at a temperature 166.3 K within a carbon nano-channel with width of 8 nm and a characteristic flow velocity of 50 m/s [HVBH09]. 22
Figure 8: Example of visual artefacts from using a texture-based rendering approach for particle data sets: precision of normal textures insufficient; Image was rendered by TexMol [BDST04] (image from [Rei08]). 28
Figure 9: Simplified process of GPU-based ray casting: The blue stages are part of the rendering hardware, while the green stages are operations of the ray

casting. First the size of the glyph's silhouette is estimated in vertex processing to span an image-space quad. For the fragments of this quad the ray casting is performed. 29
Figure 10: Evaluation of the horizontal image-space extent of a sphere at position $\boldsymbol{p}$ with radius r; The image-space extent as seen from camera $\boldsymbol{c}$ is given by the points $\boldsymbol{b}1$ and $\boldsymbol{b}2$. 32
Figure 11: Definition of three main axis vectors of an objects-space bounding box for re-construction in image space 33
Figure 12: Ball-and-Stick representation of the 2ONV Protein. The used graphical primitives are spheres and cylinders. 34
Figure 13: Determining the spanning points for the image-space footprint of a cylinder; Left: near or far points are chosen based on the cameras x coordinate only. Right: The spanning points $\boldsymbol{b}1$, $\boldsymbol{b}2$, $\boldsymbol{b}3$, and $\boldsymbol{b}4$ are calculated based on the tangent planes (blue and green) on the cylinder going through the camera position and touching the cylinder at the lines $l1$ (blue tangent plane) and $l2$ (green tangent plane). 35
Figure 14: upload performance (in fps) for rendering 106 particles as points (upper diagram) or ray cast as spheres (lower diagram) 41
Figure 15: upload performance (in fps) for rendering 107 particles as points (left diagram) or ray cast as spheres (right diagram) 42
Figure 16: upload performance of quantised positional data in fps. VBO static values (no upload) included for rendering performance reference of the corresponding hardware 45
Figure 17: Read performance from secondary storage; all values are median values over 12 measurements and are given in MB/s 47
Figure 18: Schematic process steps of a two-stage culling method for particle-based data. Generating the depth mip-map in stage 3 is required for fine-grain culling during stage 4 and hides the latencies introduced by the occlusion queries issued in stage 2. 50
Figure 19: Details on the stages of our method: 1. Initialization of the depth buffer with known occluders from the previously rendered frame; 2. Start of HWOQs for all grid cells by testing their bounding boxes; 3. Generation of HZB; 4.1. Collection of results of HWOQs, update of the list of visible cells, and rendering of visible glyphs. Stages 1 and 4.1 can output ray cast glyphs or single flat-shaded fragments if the glyphs become too small in image-space. Stage 4.2 implements deferred shading and is described in detail in chapter 2.4.1. Note that the rendering in stage 1 initializes the depth buffer with a conservative depth splat for the HZB, as well as for subsequent render passes. 52
Figure 20: MD simulation D3 of crack propagation in solid material; 44.6 million atoms 54
Figure 21: MD simulation D4 of laser ablation; 48 million atoms 54

Figure 22: The rendering performance results from Table 8 without any culling (no cull.), cell-level culling only (cell cull.), vertex-level culling only (vertex cull.), both culling techniques together (both cull.), and both culling techniques and the deferred shading pass (def. shad.). Using both culling techniques together results in the best performance. For simple point representations the effect of the vertex-level culling is negligible. This is due to a shift of the limiting bottleneck from rendering to data transfer. The overhead of the deferred shading pass is very small. Note that the bars of data set D1 are truncated to keep the focus on the values of the larger and more interesting data sets. 56
Figure 23: Comparison between local lighting and deferred shading. Left: straight-forward ray casting of spheres. Middle: 8 • 8 super-sampling of ray cast spheres. Right: deferred shading with estimated normal vectors. 59
Figure 24: MD simulation of 2 million molecules forming a liquid layer of argon in vacuum, which gets ripped apart by its vapour pressure. The time frames shown are 5 (a)(d), 15 (b)(e), and 30 (c)(f). The lower images (d)(e)(f) show straight-forward ray casting of spheres for the individual molecules. The upper images (a)(b)(c) show the same rendering enhanced with OSAO. Especially the break-up of the structure in time frames 5 and 15 is more clearly visible using OSAO. 60
Figure 25: Maltoporin protein renderings (PDB-ID: 1AF6, 10 000 atoms); Left: the presented OSAO method; Right: depth darkening [LCD06]; Depth darkening emphasizes the three channels but is not able to extract the more shallow structures on top of the protein. 62
Figure 26: Zoomed-in view; left: local lighting and Phong shading only. Since the spheres are overlapping the depth structure is not clear, especially for the marked sphere. Middle: the ambient occlusion factors; right: the final image; The position of the marked sphere is now clearly behind all other spheres. 63
Figure 27: Interpolation of the ambient occlusion factors is based on the surface normal. The values along the main axes (arrows) are interpolated with the squares of the components of the normal vector. In the general case, all six values are thus required. 67
Figure 28: Comparison of image quality of the vertex-shader-based OSAO evaluation (left) and the fragment-shader-based evaluation (right). The fragment-shader-based approach better captures the neighbourhood information, as can be seen by the shadow of the small floating cluster at the top right, and the dark gap between the different atom layers at the bottom left. 70
Figure 29: Comparison of different Shading techniques: top-left: naïve ray casting, top-right: ray tracing with ambient occlusion; bottom-left: SSAO; bottom-right: the presented OSAO approach. Data set: Cowpea chlorotic mottle virus with 220,000 atoms. 71

Figure 30: Comparison of different Shading techniques: top-left: naïve ray casting, top-right: ray tracing with ambient occlusion; bottom-left: SSAO; bottom-right: the presented OSAO approach. Data set: laser ablation with 560,000 particles. 72
Figure 31: The nickel-alumina data set (NiAl) exposes visual artefacts. While the overall structure is clearly emphasized (cf. zoomed-in views on the left), the flat planes formed by the crystal lattice are not aligned with the particle density volume. The right zoomed-in view shows the artefacts with exaggerated density volume influence to make them better perceptible. 73
Figure 32: A laser ablation data set of 11.8 million particles; the shape of the crater's brim is not emphasized because the OSAO only uses short-range neighbourhood information to determine the occlusion factors. 74
Figure 33: Simple compound glyphs, which can be ray cast in a single optimized GPU shader; right: arrow glyphs consisting of cylinders and cones, colour-coded based on their direction; left: dipole-glyph consisting of two spheres representing mass centers and one two-coloured cylinder representing a magnetic dipole moment [RE05]. 75
Figure 34: A typical real-world particle dataset from the field of molecular dynamics, containing a mixture of ethanol and R227ea, 1M molecules altogether, represented by GPU-based glyphs composited from 8.25M graphical primitives. 76
Figure 35: Two complex molecules modelled with spheres and cylinders. Left: an ethanol molecule with the orange stick representing the carbon backbone and three spheres showing point charges; Right: an R227ea molecule with a blue stick representation of the carbon and fluorine atoms and a bar magnet showing a directed charge. 77
Figure 36: Two crystallites used to model the porous media; Left: rather uniform crystallite with 18 faces; Right: crystallite with 50 randomly placed faces used for performance measurements only. Orange lines show the face normal vectors defining the tangent planes. 79
Figure 37: The principle of ray casting convex polyhedron glyphs (in 2D convex polygons). Viewing ray $r1$hits tangent plane of normal $\boldsymbol{n}1$at point $\boldsymbol{p}1$. Hit point $\boldsymbol{p}2$ is the farthest front face hit point and thus the correct point. Viewing ray $r2$ would choose $\boldsymbol{p}3$this way. However, the front-most back face hit point $\boldsymbol{p}4$ with plane of normal $\boldsymbol{n}3$ is closer to the viewer. Thus viewing ray $r2$ does not hit the polyhedron (polygon) at all. 81
Figure 38: Rendering performance of all methods in fps for tetrahedral-shaped crystallites (dashed lines) and 20-faced crystallites (solid lines); Geometry shader is fastest for 4-faced crystallite (orange line).Most methods decrease linearly with the number of particles, except for

when rendering very few particles. Ray casting results (blue lines) in better than linear scaling and thus are beneficial for large data sets......83
Figure 39: Some data sets used for the performance measurements (cf. Table 12) with (from left to right) 105, 106, 107, and 108 particles. The used crystallite types always have 20 faces......85
Figure 40: Molecular clusters are normally represented using a simple color coding (left). Using ellipsoids allows easier perceiving the shape of the clusters (right)......89
Figure 41: Point-based visualisation of an *argon* nucleation simulation data set. The left image highlights molecular clusters using different colours; the middle image uses ellipsoids and the right one uses metaballs. The metaball visualisation shows closed surfaces of molecule clusters while not exaggerating their size. 90
Figure 42: The density function used for the metaballs evaluation presented in this section; An isosurface of value 1 will yield the same impression as ray casting spheres if all particles are sufficiently far away from each other. 91
Figure 43: Occluded fragments hamper the image space computation of metaballs: the dashed influence sphere of P3 is completely occluded by P1. Therefore, it is not possible to accumulate the density function correctly in image space. As the contribution of P3 is completely missing, the dotted metaball surface cannot be found and P2 appears as normal sphere......92
Figure 44: Additional data sets tested with the metaball technique: left: disulfide bond formation protein *1A2J* from the protein data base [BWF+00]; right: a nucleation simulation of supersaturated *ethane*......94
Figure 45: The different rendering passes of the *Walking Depth Plane* approach, together with the two frame buffer objects involved......95
Figure 46: Metaball isosurface approximation in *Sim-3* data set after different number of iterations. Lower right image shows final output image of the *Walking Depth Plane* approach. The other images show the content of the λ-buffer after different numbers of iterations (from upper left to lower right): directly after initialisation, after 5, 10, 15, and 31 (final result) iterations......97
Figure 47: The metaball of the data set *Sim-3* after 5, 10, and 30 iterations (from left to right). The red colour channel encodes the finished pixels black (showing the isosurface in cyan) and unfinished pixels red......97
Figure 48: Visual artefacts: left: under-sampling artefacts occur when rendering the ethane data sets with the *Moving Depth Plane* and using too few steps; right: Missed surface parts (arrow) due to under-sampling by the *Vicinity Texture* approach......98
Figure 49: Flow chart of the sliced ray casting method; After initial ray setup, multiple rendering sub-passes of sampling position update, density

reconstruction through splatting and volume ray-casting are performed to generate the final image. 99
Figure 50: Perspective correction is necessary when projecting splats from object space into image space. Otherwise the splats are not fully represented in image space. 100
Figure 51: Two example visualizations created with the density volume isosurface approach; Left: Visualizations of the oxidoreductase protein with ~75,000 atoms (~10,000 amino acids). The lower left sub-image shows a cartoon representation of the secondary structure. The lower right sub-image shows the described surface volume rendering while the top sub-image shows a combination of both. Right: Laser ablation simulation showing a bulge on a metal block. The cutaway of the bulge reveals the density distribution inside. 101
Figure 52: Close-ups of a small region of the laser ablation data set shown in Figure 51: volume construction in (left) object space or (right) image space. Ridges, grooves, highlights, and small features are more distinct in image space. 102
Figure 53: 2D-schematic of the molecular surface definitions. The rolling probe traces out the SAS and the SES. 103
Figure 54: Illustration of the Contour-Buildup algorithm to compute the SES for a small protein (3NVF). The input is a set of intersecting atoms whose radii are expanded from VdW surface by the radius of a probe sphere to form the SAS (left). The algorithm extracts the contours from which the surface patches of the SES can be derived (middle). The right image shows the final rendering of the SES, coloured by element. 104
Figure 55: The processing work flow of the parallel GPU Contour-Building algorithm. Each blue box represents a CUDA compute kernel. Note that there is only a single data upload (green box, representing a VBO) for computation as well as rendering. 106
Figure 56: The data sets presented in Table 15; from left to right: 1OGZ, 1VIS, and 1AF6 107
Figure 57: Overview of the method to extract dislocations and stacking faults. From left to right: first the atoms are classified based on their 12-neighbourhood. The neighbourhood graph is contracted and simplified to get the topological structure of the dislocation network. This graph is then used to segment the atom data set. Finally tubes and planes are fitted into the segmented atom positions. 109
Figure 58: Graph contraction; left initial neighbourhood graph, right after contraction. Only Edges from Ed are shown. 110
Figure 59: The mass-spring system contracts the neighbourhood graph, but the distance d is too large for the edge-collapsing threshold and the three-edge cycles become stable. 111

Figure 60: Four nodes and edges in the graph forming a stable diamond which represents a single dislocation. Yellow and red atoms are incorrectly separated. All four nodes are combined to a single one. 111
Figure 61: Left: the graph after all simplifications; each edge connects a junction node to a dislocation node. Right: the final result; Junctions are shown as cyan spheres, dislocations as blue spline-tubes, and stacking faults as orange planes. 112
Figure 62: A problematic situation for the stacking fault segmentation using region growing; left: the original atom data; in the neighbourhood graph (middle image) atoms from different stacking fault segments are connected (orange lines between points from both stacking faults visible as grey areas) because of the acute angle under which the two stacking faults meet. The right image shows the final (correct) segmentation. 112
Figure 63: Extracted crystal defects in a time-dependent data set at time frames 10, 250, 350, 400, 500, and 595 113
Figure 64: Data set with two layers (upper layer Ni3Al; lower layer Ni) being pulled apart 114
Figure 65: Data set with only Ni3Al pulled in one direction 114
Figure 66: Assignment of points to sites in one time step of the laser ablation data set. Sites are shown as large spheres. The assignment of the original particle data is represented by the colour. 116
Figure 67: Results for the laser ablation data set (3750 sites for 562,500 points) showing spline tubes as path lines over 400 time steps. Left: capacity constrained to 150, middle: capacity 100, 200, right: capacity 20, 300...117
Figure 68: Simulation of a collision of two liquid droplets (methane and ethane) in vacuum for the first 500 time steps. Top four images: path line visualization of sites generated with a time limit of 200 seconds per time frame and the capacity constraints (from left to right) 489 points, 350, 650, and 16, 2048. Right image of middle row: the representation generated with a constraint of 16, 2048 (same as left, middle image) but without the time limit of 200 seconds. Bottom: two snapshots of the simulation: time frames 200 and 500. Ethane is shown in red and methane is shown in green. 122
Figure 69: Nucleation simulation of CO_2. Left: the trajectories of the sites resemble homogeneous, chaotic movement (actually present in the vapour phase of the data), but fails to show any clear clustering of the nuclei (constraints [256, 2048]). Right: applying domain-knowledge of the data sets and setting the site constraint boundaries low enough to capture the atom cluster size ([4, 64]) reveals the atom clusters in the site distribution clearly but results in too many sites for the path line approach. 123

Figure 70: The original path lines are smoothed (right) to emphasise the principal directions of motion normally obscured by the molecule's chaotic small-scale fluctuations (left) 125
Figure 71: Clustering of the path lines p_i: vertices w_{it} (grey) along a path line with low velocity values form spherical clusters $c_{i,n}$(marked in light blue). Those clusters overlapping define a new major cluster $c_k{}'$ (green) connecting the individual path lines. 126
Figure 72: A small subset of a principal path line network before (left) and after (right) adjusting the Bézier curve control points based on neighbouring curves (image from [Bid10]) 127
Figure 73: Extraction of principal path lines network. Left: all path lines after smoothing; Centre: the vertices of all path lines, coloured based on the detected clusters of slower motion. Positions of conserved water found through crystallography correspond to the larger clusters. Right: the extracted principal paths 127
Figure 74: Structure of principal solvent paths in context of a cartoon representation of a surrounding protein (red). The image shows the extracted main solvent paths. The time frame is mapped to a colour gradient from yellow (start of the trajectory) to blue (end of the trajectory). 128
Figure 75: Clusters with colours between yellow and red are decaying (left), while clusters with colours between yellow and green are growing (right). Evaporating molecular cluster (in the middle of the left image) leaving a flow group (blue arrow, centre image) containing almost all molecules of the evaporated cluster (indicated by the similar size), moving slowly and forming a new cluster (right image). This behaviour hints from a cluster detection problem 130
Figure 76: The schematic view of the molecule cluster evolution with two selected clusters (red) and all corresponding flow groups connecting to other clusters (black) shows the 10,000 methane nucleation dataset with geometrical cluster detection. The colour of the flow groups are based on the IDs e and v of the clusters they are leaving and joining to allow for visual grouping. 132
Figure 77: Schematic views of two clustering algorithms applied to the 50,000 methane nucleation dataset. The top images show the results of a pure geometric clustering, and the lower images show the results of a clustering based on energy levels. The images below the diagrams show zoomed-in views of the selected cluster (marked red in the diagrams) at different time frames. The geometrical clustering detects large and unstable clusters. The selected cluster is temporarily split into two. The thick green flow groups and the corresponding dent in the lower cluster line indicate a detection problem. The energy-based clustering (right) is more stable in this respect but results in smaller clusters. 133

Figure 78: Schematic view of the R-152a nucleation simulation (Figure 4; right). Almost all molecules are very quickly clustered in many rather small and stable clusters. Greater changes only happen when two clusters merge, which is clearly indicated by the corresponding thick flow groups in the schematic view. 134
Figure 79: Left: the classic bathtub curve as known from traditional reliability analysis. The integral of the curve until a given point in time relates to the effort invested. Right: the bathtub curve for software exhibits two main differences [Rel96]: software does not decay and thus cannot wear out. However, it will be updated several times over its useful life, which will always add new bugs. In the long run, functionality will not fit the original design and thus deteriorate quality, increasing the failure probability with each update. 141
Figure 80: Bathtub curves for different kinds of software developed in the context of research. The blue curve represents the base workload required for development of framework aspects, e.g. data set loading or the main loop. The green curve (identical in all three diagrams) adds the workload required for the algorithmic method development for each publication. 142
Figure 81: Schematic illustration of the proposed development process: Instead of monolithic prototype implementation reusable classes are identified and integrated into a local repository. Only for these classes additional effort is required guaranteeing high source code quality, like thorough documentation. 146
Figure 82: Module structure of the VIS*lib*. The modules, which are implemented as static libraries, are organized in three tiers building-up on each other. 147
Figure 83: The three phases of the process for *visualization systems engineering*. Subtask sequence is shown by the horizontal position; vertical ordering is arbitrary and denotes possible parallelism. 150
Figure 84: Sequence diagram of a simple rendering module; the resulting image is cached in a FBO and only re-rendered if the data changes, which is detected by testing the data hash value provided by the source module. 153
Figure 85: Example of a MegaMol module graph. The entry module View3D is connected to a Mux4Renderer3D which combines the results of four different rendering modules. Utimately data is only loaded from and then provided by the Module IMDAtomData loading a particle data file of the corresponding format. Other modules enrich the corresponding view of the data (e.g. DirPartColModulate add a new colour channel) or derive new data (e.g. IsoSurface). 154
Figure 86: Different GUI frontends of MegaMol: *top-left*: the current MegaMol Console frontend for debugging and Powerwall support; *top-centre*: a specialized application for teaching, in which the mapping operations

scripts can be written in C#; *top-right*: an older GUI of MegaMol (V.:0.2), written with Windows Forms; bottom-left: a GUI written in Qt for the application of cavity analysis [KFR+11]; *bottom-right*: A GUI supporting multiple coordinated views, also written with Qt..157

List of Tables

Table 1: Performance (fps) of ray casting cylinders, depending on the chosen bounding geometry approximation method: point-based, quads, or using the geometry shader. Data set contains 500,000 non-overlapping cylinders with length-radius ratio of 2:1. 36
Table 2: Explanation of the different uploading mechanisms used in the performance tests........ 39
Table 3: upload performance (in fps) for rendering 106 particles as points (upper part) or ray castes as spheres (lower part) 40
Table 4: upload performance (in fps) for rendering 107 particles as points (upper part) or ray cast as spheres (lower part)........ 42
Table 5: upload performance values (fps) for uploading quantised positional data. 44
Table 6: Read performance from secondary storage; all values are median values over 12 measurements and are given in MB/s. 48
Table 7: Sizes and descriptions of the example data sets: D1 – D4 have been created by molecular dynamics simulations, D5 is an artificial data set and has been created using a statistical distribution. 53
Table 8: Performance measurements of the two-stage occlusion culling method; All numbers indicate FPS if not otherwise noted. Different views were used such that glyphs are large enough in image space and rendered as ray cast spheres (S-*), or splatted as simple points (P-*). The viewing directions and distances were changed to obtain a best (*-Best) and worst (*-Worst) case for the culling algorithms (i.e. maximum/minimum number of grid cells occluded). The last two columns show statistics for the case that both culling levels are active: the percentage of the grid cells that are visible and the number of glyphs that are actually ray cast after the vertex culling stage. Column 3 (culling: none) shows the baseline performance for the un-optimized approach. Columns 6 and 7 demonstrate the impact of caching. 55
Table 9: Data sets used to test the presented OSAO method........ 68
Table 10: The performance results of the presented OSAO method, including generation of the particle density volume and rendering. All values are in *frames per second*. The rendering mode *no AO* refers to a renderer without any ambient occlusion calculations (i.e. no density volume is generated). The rendering mode *none* describes the test without rendering where only the density volume is computed and transferred to the graphics card. 69
Table 11: Number of triangles and vertices per crystallite for a given number of faces; the numbers vary slightly for the different crystallite templates due to the different plane cutting conditions. The two numbers of vertices show the number of *unique* vertices, required when storing the mesh

data in VBOs, and the number of vertices needed to be *drawn* based on the number of triangles (relevant for the geometry shader approach). 80
Table 12: Rendering performance values in fps of the different rendering techniques; *#Faces* are the number of crystallite faces (not triangles). The rendering techniques are: VBO – VBO-based rendering, Inst – hardware-supported instancing, GPUGeom – geometry-shader based, and Raycast – point-based ray casting 82
Table 13: Performance values of the *Vicinity-Texture* approach: The processing time specifies the time needed to build up the required data structure texture (view-independent) using a plane-sweep algorithm; Number of potential metaball members in the different data sets: Non-empty groups are the spheres which have at least one neighbour which is close enough to form a metaball with. Spheres which can never form a metaball are counted as empty groups. The size column shows the minimum, average and maximum vicinity group size in the data set. . 93
Table 14: Performance values of the *Walking Depth Plane* method. The maximum number of iterations is changed for comparability or for artefact-free visualization. 98
Table 15: Performance measurements for the CUDA implementation of the parallel Contour-Buildup algorithm; *CB* denotes the execution time of the Contour-Buildup algorithm (including data transfer). *VBO+SH* denotes the time for writing VBOs for rendering and the handling of singularities. The column labelled *fps* states the overall performance in frames per second (i.e. data transfer, Contour-Buildup computation and rendering of the SES). 106
Table 16: Computation times and quality parameters for varying numbers of sites using the constant and the camel density functions in two dimensions (cf. [FSG+11]). All results were obtained by averaging optimization runs from 10 independently generated sets of sites obtained via rejection sampling. The number of discrete sample points per site was 4096 except where otherwise noted. 120
Table 17: Performance of the CUDA implementation of LCCVDs for different data sets in seconds 121

Bibliography

[AM92] P. K. Agarwal and J. Matausek, "Relative neighborhood graphs in three dimensions," in *Proceedings of the 3rd annual ACM/SIAM symposium on Discrete algorithms*, 1992, pp. 58–65.

[Ami] "Amira," http://www.amira.com/ (last visited: 17.01.2012).

[AR99] J. L. Arjona and G. Roberts, "The Layers of Change in Software Architecture," in *Software Architecture, 1999. Proceedings. Working IEEE/IFIP Conference on*, 1999.

[ATC+08] O. K.-C. Au, C.-L. Tai, H.-K. Chu, D. Cohen-Or, and T.-Y. Lee, "Skeleton extraction by mesh contraction," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 1–10, 2008.

[AVS] "AVS," http://www.avs.com/ (last visited: 17.01.2012).

[BAK+98] V. Bulatov, F. F. Abraham, L. Kubin, B. Devincre, and S. Yip, "Connecting atomistic and mesoscale simulations of crystal plasticity," *Nature*, vol. 391, pp. 669–672, 1998.

[Bau93] F. L. Bauer, "Software Engineering - wie es begann," *Informatik Spektrum*, pp. 259–260, 1993.

[BBD00] P. S. Bradley, K. P. Bennett, and A. Demiriz, "Constrained k-means clustering," MSR-TR-2000-65, Microsoft Research, Tech. Rep., 2000.

[BC06] V. Bulatov and W. Cai, *Computer Simulation of Dislocations*. Oxford University Press: Oxford, New York, 2006.

[BCF+04] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis, "Scalable Line Dynamics in ParaDiS," in *Proceedings of the ACM/IEEE conference on Supercomputing*, 2004, p. 19.

[BDST04] C. Bajaj, P. Djeu, V. Siddavanahalli, and A. Thane, "TexMol: Interactive Visual Exploration of Large Flexible Multi-Component Molecular Complexes," in *Proceedings of IEEE Visualization 2004*, 2004, pp. 243–250.

[BDW08] S. Basu, I. Davidson, and K. Wagstaff, *Constrained Clustering: Advances in Algorithms, Theory, and Applications*. Chapman & Hall/CRC, 2008.

[BFH04] I. Buck, K. Fatahalian, and P. Hanrahan, "Gpubench: Evaluating gpu performance for numerical and scientific applications." in *Poster Session at GP2 Workshop on General Purpose Computing on Graphics Processors*, 2004, http://gpubench.sourceforge.net/.

[BG02] A. Banerjee and J. Ghosh, "On scaling up balanced clustering algorithms," in *Proceedings of the SIAM International Conference on Data Mining*, 2002, pp. 333–349.

[BGB+08] K. Bidmon, S. Grottel, F. Bös, J. Pleiss, and T. Ertl, "Visual Abstractions of Solvent Pathlines near Protein Cavities," in *Computer Graphics Forum*, vol. 27, no. 3, 2008, pp. 935–942.

[BH08] M. Balzer and D. Heck, "Capacity-Constrained Voronoi Diagrams in Finite Spaces," in *Proceedings of the 5th Annual International Symposi-*

um on Voronoi Diagrams in Science and Engineering, no. 4(2), 2008, pp. 44–56.

[BHT⁺06] V. V. Bulatov, L. L. Hsiung, M. Tang, A. Arsenlis, M. C. Bartelt, W. Cai, J. N. Florando, M. Hiratani, M. Rhee1, G. Hommes, T. G. Pierce, and T. D. de la Rubia, "Dislocation multi-junctions and strain hardening," *Nature*, vol. 440, pp. 1174–1178, 2006.

[BHZK05] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt, "High-quality surface splatting on today's GPUs," in *Proceedings of Symposium on Point-Based Graphics 2005*, 2005, pp. 17–141.

[Bid10] K. Bidmon, "Processing of Meshes and Geometry for Visualization Applications," Dissertation, Visualisierungsinstitut der Universität Stuttgart, Stuttgart, Deutschland, 2010.

[BK03] M. Botsch and L. Kobbelt, "High-Quality Point-Based Rendering on Modern GPUs," in *Pacific Graphics 2003*, 2003, pp. 335–343.

[Bli77] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *SIGGRAPH Computer Graphics*, vol. 11, pp. 192–198, July 1977.

[Bli82] ——, "A Generalization of Algebraic Surface Drawing," *ACM Transactions on Graphics*, vol. 1, pp. 235–256, 1982.

[Boo04] G. Booch, *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., 2004.

[BS09] L. Bavoil and M. Sainz, "Image-Space Horizon-Based Ambient Occlusion," in *Shader X7: Advanced Rendering Techniques*, W. Engel, Ed. Charles River Media, 2009, pp. 425–444.

[BSD09] M. Balzer, T. Schlömer, and O. Deussen, "Capacity-Constrained Point Distributions: A Variant of Lloyd's Method," *ACM Transactions on Graphics*, vol. 28, no. 3, pp. 86–94, 2009.

[BSK04] M. Botsch, M. Spernat, and L. Kobbelt, "Phong Splatting," in *Proceedings of Symposium on Point-Based Graphics 2004*, 2004, pp. 25–32.

[Bun05] M. Bunnell, "Dynamic Ambient Occlusion and Indirect Lighting," in *GPU Gems 2*, M. Pharr and R. Fernando, Eds. Addison-Wesley, 2005, ch. 14, pp. 223–233.

[BW03] J. Bittner and P. Wonka, "Visibility in Computer Graphics," *Environment and Planning B: Planning and Design*, vol. 30, no. 5, pp. 729–756, 2003.

[BW08] S. Bachthaler and D. Weiskopf, "Continuous Scatterplots," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1428–1435, 2008.

[BWF⁺00] H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, and P. Bourne, "The Protein Data Bank," *Nucleic Acids Research*, vol. 28, pp. 235–242, 2000.

[BWK02] M. Botsch, A. Wiratanaya, and L. Kobbelt, "Efficient high quality rendering of point sampled geometry," in *Proceedings of the 13th Eurographics workshop on Rendering*, 2002, pp. 53–64.

[BWPP04] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, “Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful,” *Computer Graphics Forum*, vol. 23, no. 3, pp. 615–624, 2004.

[BWT06] C. A. Bottoms, T. A. White, and J. J. Tanner, “Exploring structurally conserved solvent sites in protein families,” *Proteins*, vol. 64, no. 2, pp. 404–421, 2006.

[CCD+05] D. Case, T. Cheatham, T. Darden, H. Gohlke, R. Luo, K.M. Merz, Jr., A. Onufriev, C. Simmerling, B. Wang, and R. Woods, “The amber biomolecular simulation programs,” *Journal of Computational Chemistry*, vol. 26, pp. 1668–1688, 2005.

[CE97] M. Chuah and S. Eick, “Glyphs for software visualization,” *Proceedings of the Fifth Iternational Workshop on Program Comprehension, 1997*, pp. 183–191, Mar 1997.

[CLM08] M. Chavent, B. Levy, and B. Maigret, “MetaMol: High-quality visualization of molecular skin surface,” *Journal of Molecular Graphics and Modelling*, vol. 27, no. 2, pp. 209–216, 2008.

[CMS99] S. K. Card, J. D. Mackinlay, and B. Shneiderman, *Readings in information visualization: Using vision to think*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[CMS07] N. D. Cornea, P. Min, and D. Silver, “Curve-Skeleton Properties, Applications, and Algorithms,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 3, pp. 530–548, 2007.

[COCSD03] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand, “A Survey of Visibility for Walkthrough Applications,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 09, no. 3, pp. 412–431, 2003.

[Con83] M. L. Connolly, “Analytical Molecular Surface Calculation,” *Journal of Applied Crystallography*, vol. 16, pp. 548–558, 1983.

[Con85] ——, “Molecular surface Triangulation,” *Journal of Applied Crystallography*, vol. 18, no. 6, pp. 499–505, Dec 1985.

[Cro77] F. C. Crow, “Shadow Algorithms for Computer Graphics,” *ACM SIGGRAPH*, vol. 11, pp. 242–248, 1977.

[CSA03] H. Carr, J. Snoeyink, and U. Axen, “Computing contour trees in all dimensions,” *Computational Geometry: Theory and Application*, vol. 24, no. 2, pp. 75–94, 2003.

[CSI09] D. Cha, S. Son, and I. Ihm, “GPU-Assisted High Quality Particle Rendering,” in *Eurographics Symposium on Rendering*, 2009, pp. 1247–1255.

[CSSS06] H. Czichos, T. Saito, L. R. Smith, and L. Smith, *Springer Handbook of Material Measurement Methods*. Springer, 2006.

[cud11] “CUDA Data Parallel Primitives Library,” April 2011, http://gpgpu.org/developer/cudpp/ (last visited: 17.01.2012).

[CY79] L. L. Constantine and E. Yourdon, *Structured Design*. Prentice-Hall, 1979.

[Déc05] X. Décoret, “N-Buffers for Efficient Depth Map Query,” *Computer Graphics Forum*, vol. 24, no. 3, pp. 393–400, 2005.

[DCH88] R. A. Drebin, L. Carpenter, and P. Hanrahan, “Volume rendering,” in *ACM SIGGRAPH 1988*, 1988, pp. 65–74.

[DeL02] W. L. DeLano, “The PyMOL molecular graphics system,” 2002, http://www.pymol.org/ (last visited: 17.01.2012).

[DFG] “Dynamic simulation of systems with large particle numbers,” http://www.dfg.de/en/research_funding/programmes/list/projectdetails/index.jsp?id=17546514 (last visited: 19.01.2012); http://www.sfb716.uni-stuttgart.de/ (last visited: 17.01.2012).

[DH73] R. O. Duda and P. E. Hart, *Pattern classification and scene analysis*. Wiley New York, 1973.

[DKL98] E. B. Dam, M. Koch, and M. Lillholm, “Quaternions, Interpolation and Animation,” Department of Computer Science, University of Copenhagen, Tech. Rep., 1998.

[dTLP07] R. de Toledo, B. Lévy, and J.-C. Paul, “Iterative Methods for Visualization of Implicit Surfaces on GPU,” in *International Symposium on Visual Computing*, Nov 2007, pp. 598–609.

[Dub05] P. Dubois, “Maintaining Correctness in Scientific Programs,” *Computing in Science and Engineering*, vol. 7, no. 3, pp. 80–85, May-June 2005.

[ED06] M. Eissele and J. Diepstraten, “GPU Performance of DirectX 9 Per-Fragment Operations Revisited,” in *Shader X4: Advanced Rendering with DirectX and OpenGL*, Wolfgang Engel, Ed. Charles River Media, 2006, pp. 541–560.

[ED09] T. Engelhardt and C. Dachsbacher, “Granular Visibility Queries,” in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2009, pp. 161–167.

[EHM+08] H. Edelsbrunner, J. Harer, A. Mascarenhas, V. Pascucci, and J. Snoeyink, “Time-varying Reeb graphs for continuous space–time data,” *Computational Geometry: Theory and Application*, vol. 41, no. 3, pp. 149–166, 2008.

[Ei90] I. O. Electrical and E. E. (ieee), *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[EM94] H. Edelsbrunner and E. P. Mücke, “Three-dimensional alpha shapes,” *ACM Transactions on Graphics*, vol. 13, no. 1, pp. 43–72, 1994.

[Eve01] C. Everitt, “Interactive Order-Independent Transparency,” NVidia, Tech. Rep., 2001.

[Ewa21] P. P. Ewald, “Die Berechnung optischer und elektrostatischer Gitterpotentiale,” *Annalen der Physik*, vol. 369, no. 3, pp. 253–287, 1921.

[FC08] M. Fox and S. Compton, “Ambient Occlusive Crease Shading,” *Game Developer Magazine*, pp. 19–23, Mar 2008.

[FGE10] M. Falk, S. Grottel, and T. Ertl, “Interactive Image-Space Volume Visualization for Dynamic Particle Simulations,” in *Proceedings of The Annual SIGRAD Conference*. Linköping University Electronic Press, 2010, pp. 35–43.

[For04] I. Ford, “Statistical mechanics of nucleation: a review,” in *Proceedings of the Institution of Mechanical Engineers, Part C, Journal of Mechanical Engineering Science*, vol. 218, August 2004, pp. 883–899.

[FRE12] S. Frey, G. Reina, and T. Ertl, “SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms,” in *Proceedings of the 2012 20th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP ’12, 2012, pp. 399–406.

[FRK+98] J. A. Fisk, M. M. Rudek, J. L. Katz, D. Beiersdorf, and H. Uchtmann, “The homogeneous nucleation of cesium vapor,” *Atmospheric Research*, vol. 46, pp. 211–222, 1998.

[FSG+11] S. Frey, T. Schloemer, S. Grottel, C. Dachsbacher, O. Deussen, and T. Ertl, “Loose Capacity-Constrained Representatives for the Qualitative Visual Analysis in Molecular Dynamics,” in *Proceedings of IEEE Pacific Visualization Symposium*, 2011, pp. 51–58.

[GB78] J. Greer and B. L. Bush, “Macromolecular shape and surface maps by solvent exclusion,” in *Proceedings of the National Academy of Science*, Jan 1978, pp. 303–307.

[GBK06] M. Guthe, A. Balázs, and R. Klein, “Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries,” in *Eurographics Symposium on Rendering*, T. Akenine-Möller and W. Heidrich, Eds., June 2006, pp. 207–214.

[GBP04] G. Guennebaud, L. Barthe, and M. Paulin, “Deferred Splatting,” *Computer Graphics Forum*, vol. 23, no. 3, pp. 653–660, septembre 2004.

[GDCE09] S. Grottel, C. A. Dietrich, J. L. D. Comba, and T. Ertl, *Topological Methods in Data Analysis and Visualization*. Springer, 2009, ch. Topological Extraction and Tracking of Defects in Crystal Structures, pp. 167–178.

[Geo] “GLSL geometry shader 4 extension specification,” http://developer.download.nvidia.com/opengl/specs/GL_EXT_geometry_shader4.txt (last visited: 17.01.2012).

[GKM93] N. Greene, M. Kass, and G. Miller, “Hierarchical Z-Buffer Visibility,” in *ACM SIGGRAPH 1993*, 1993, pp. 231–238.

[GKSE12] S. Grottel, M. Krone, K. Scharnowski, and T. Ertl, “Object-Space Ambient Occlusion for Molecular Dynamics,” in *Proceedings of IEEE Pacific Visualization Symposium*, 2012, pp. 209–216.

[GKZ07] M. Griebel, S. Knapek, and G. Zumbusch, *Numerical Simulation in Molecular Dynamics*. Berlin, Heidelberg: Springer, 2007.

[GM77] R. Gingold and J. Monaghan, “Smoothed Particle Hydrodynamics: Theory and Application to Non-spherical Stars,” *Monthly Notices Royal Astronomical Society*, vol. 181, pp. 375–389, 1977.

[GMRE09] S. Grottel, C. Müller, G. Reina, and T. Ertl, “A Lean Process Model using In-House Source Code Libraries for Efficient Development of Visualization Applications,” in *IEEE VisWeek - Workshop Refactoring Visualization from Experience*, 2009.

[GNP+06] A. Gyulassy, V. Natarajan, V. Pascucci, P.-T. Bremer, and B. Hamann, "A Topological Approach to Simplification of Three-Dimensional Scalar Functions," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 4, pp. 474–484, 2006.

[GP07] M. Gross and H. Pfister, Eds., *Point-based Graphics*. Morgan Kaufmann, 2007.

[GRDE10] S. Grottel, G. Reina, C. Dachsbacher, and T. Ertl, "Coherent Culling and Shading for Large Molecular Dynamics Visualization," in *Computer Graphics Forum*, vol. 29, 2010, pp. 953–962, http://www.vis.uni-stuttgart.de/grottel/eurovis10/ (last visited 17.06.2012).

[Gre08] S. Green, "CUDA Particles," NVIDIA Corporation, Tech. Rep., 2008.

[GRE09] S. Grottel, G. Reina, and T. Ertl, "Optimized Data Transfer for Time-dependent, GPU-based Glyphs," in *Proceedings of IEEE Pacific Visualization Symposium*, 2009, pp. 65–72, http://www.vis.uni-stuttgart.de/eng/research/fields/perf/ (last visited 17.06.2012).

[GRVE07] S. Grottel, G. Reina, J. Vrabec, and T. Ertl, "Visual Verification and Analysis of Cluster Detection for Molecular Dynamics," in *IEEE Transactions on Visualization and Computer Graphics*, 2007, pp. 1624–1631.

[GRZ+10] S. Grottel, G. Reina, T. Zauner, R. Hilfer, and T. Ertl, "Particle-based Rendering for Porous Media." in *Proceedings of The Annual SIGRAD Conference*. Linköping University Electronic Press, 2010, pp. 45–51.

[Gum03] S. Gumhold, "Splatting Illuminated Ellipsoids with Depth Correction," in *Proceedings of VMV*, 2003, pp. 245–252.

[HA87] J. D. Honeycutt and H. C. Andersen, "Molecular dynamics study of melting and freezing of small Lennard-Jones clusters," *The Journal of Physical Chemistry*, vol. 91, no. 19, pp. 4950–4963, 1987.

[HCL05] J. Heer, S. K. Card, and J. A. Landay, "Prefuse: a Toolkit for Interactive Information Visualization," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ser. CHI '05. ACM, 2005, pp. 421–430.

[HDS96] W. Humphrey, A. Dalke, and K. Schulten, "VMD – Visual Molecular Dynamics," *Journal of Molecular Graphics*, vol. 14, pp. 33–38, 1996.

[HE95] D. Herbison-Evans, "Solving Quartics and Cubics for Graphics," in *Graphics Gems V*, A. W. Paeth, Ed., 1995, pp. 3–15.

[HE03] M. Hopf and T. Ertl, "Hierarchical Splatting of Scattered Data," in *Proceedings of IEEE Visualization 2003*. IEEE, 2003, pp. 57–64.

[Hel07] V. Helms, "Protein dynamics tightly connected to the dynamics of surrounding and internal water molecules," *Journal of Chemical Physics and Physical Chemistry*, vol. 8, pp. 23–33, 2007.

[Hil55] T. L. Hill, "Molecular Clusters in Imperfect Gases," *Journal of Chemical Physics*, vol. 23, pp. 617–622, April 1955.

[Hil96] R. Hilfer, "Transport and relaxation phenomena in porous media," *Advances in Chemical Physics*, vol. XCII, p. 299, 1996.

[HJ07] J. Hoberock and Y. Jia, "High-Quality Ambient Occlusion," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2007, ch. 14, pp. 257–273.

[HL92] J. P. Hirth and J. Lothe, *Theory of Dislocations*. Krieger Publishing Company, 1992.

[HLE04] M. Hopf, M. Luttenberger, and T. Ertl, "Hierarchical Splatting of Scattered 4D Data," in *IEEE Computer Graphics and Applications*, vol. 24, no. 4, 2004, pp. 64–72.

[HM05] C. Hartley and Y. Mishin, "Characterization and visualization of the lattice misfit associated with dislocation cores," *Acta Materialia*, vol. 53, pp. 1313–1321, 2005.

[HOF05] A. Halm, L. Offen, and D. Fellner, "BioBrowser: A Framework for Fast Protein Visualization," in *Computer Graphics Forum*, 2005, pp. 287–294.

[HSRG07] C. Han, B. Sun, R. Ramamoorthi, and E. Grinspun, "Frequency domain normal map filtering," *ACM Transactions on Graphics*, vol. 26, no. 3, pp. 28–40, 2007.

[HVB+08] M. Horsch, J. Vrabec, M. Bernreuther, S. Grottel, G. Reina, A. Wix, K. Schaber, and H. Hasse, "Homogeneous nucleation in supersaturated vapors of methane, ethane, and carbon dioxide predicted by brute force molecular dynamics," *Journal of Chemical Physics*, vol. 128, no. 16, p. 164510, 2008.

[HVBH09] M. Horsch, J. Vrabec, M. Bernreuther, and H. Hasse, "Poiseuille flow of liquid methane in nanoscopic graphite channels by molecular dynamics simulation," in *Proceedings of the International Symposium Turbulence, Heat and Mass Transfer*, vol. 6, 2009, pp. 89–92.

[Ins] "GL_EXT_draw_instanced Specification," http://opengl.org/registry/specs/EXT/draw_instanced.txt (last visited: 17.01.2012).

[Jal97] P. Jalote, *An Integrated Approach to Software Engineering*, 2nd ed. Springer New York, Inc., 1997.

[JAL05] B. G. James Ahrens and C. Law, *ParaView: An End-User Tool for Large Data Visualization*. Elsevier, 2005, pp. 717–731.

[JBPE99] R. P. Jensen, P. J. Bosscher, M. E. Plesha, and T. B. Edil, "DEM simulation of granular media-structure interface: effects of surface roughness and particle shape," *International Journal for Numerical and Analytical Methods in Geomechanics*, vol. 23, no. 6, pp. 531–547, 1999.

[JH04] C. Johnson and C. Hansen, *Visualization Handbook*. Orlando, FL, USA: Academic Press, Inc., 2004.

[Kaj09] V. Kajalin, "Screen-Space Ambient Occlusion," in *Shader X7: Advanced Rendering Techniques*. Charles River Media, 2009, pp. 413–424.

[KBE09] M. Krone, K. Bidmon, and T. Ertl, "Interactive Visualization of Molecular Surface Dynamics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1391–1398, 2009.

[KC05] A. Kolb and N. Cuntz, "Dynamic particle coupling for GPU-based fluid simulation," in *Proceedings of Symposium on Simulation Technique (ASIM)*, 2005, pp. 722–727.

[KE04] T. Klein and T. Ertl, "Illustrating Magnetic Field Lines using a Discrete Particle Model," in *Proceedings of VMV*, 2004, pp. 387–394.

[KFR+11] M. Krone, M. Falk, S. Rehm, J. Pleiss, and T. Ertl, "Interactive Exploration of Protein Cavities," *Computer Graphics Forum*, vol. 30, no. 3, pp. 673–682, 2011.

[KGE11] M. Krone, S. Grottel, and T. Ertl, "Parallel Contour-Buildup Algorithm for the Molecular Surface," pp. 17–22, 2011.

[KHK+09] A. Knoll, Y. Hijazi, A. Kensler, M. Schott, C. Hansen, and H. Hagen, "Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic." in *Computer Graphics Forum*, vol. 28, no. 1, 2009, pp. 26–40.

[KK86] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," in *ACM SIGGRAPH 1986*. New York, NY, USA: ACM, 1986, pp. 269–278.

[KKEM10] D. A. Keim, J. Kohlhammer, G. Ellis, and F. Mansmann, Eds., *Mastering The Information Age - Solving Problems with Visual Analytics*. Eurographics, 2010.

[KKKW05] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann, "A Particle System for Interactive Visualization of 3D Flows," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 6, pp. 744–756, 11 2005.

[Kle08] T. Klein, "Exploiting Programmable Graphics Hardware for Interactive Visualization of 3D Data Fields," Dissertation, Universität Stuttgart, Stuttgart, Deutschland, 2008.

[KLR04] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-based simulation and collision detection for large particle systems," in *Eurographics Workshop on Graphics Hardware*, 2004, pp. 123–131.

[Kol01] E. Kolatch, "Clustering algorithms for spatial databases: A survey," Tech. Rep., 2001.

[KS00] J. T. Klosowski and C. T. Silva, "The Prioritized-Layered Projection Algorithm for Visible Set Estimation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 2, pp. 108–123, 2000.

[KS09] J. Kalojanov and P. Slusallek, "A Parallel Algorithm for Construction of Uniform Grids," in *Proceedings of ACM SIGGRAPH/Eurographics High Performance Graphics*. ACM, 2009, pp. 23–28.

[KSN08] Y. Kanamori, Z. Szego, and T. Nishita, "GPU-based Fast Ray Casting for a Large Number of Metaballs," *Computer Graphics Forum*, vol. 27, no. 3, pp. 351–360, 2008.

[KSW04] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: a GPU-based particle engine," in *Eurographics Workshop on Graphics Hardware*, 2004, pp. 115–122.

[KW03] J. Krüger and R. Westermann, “Acceleration techniques for GPU-based volume rendering,” in *Proceedings IEEE Visualization 2003*, 2003, pp. 287–292.

[LB06] C. Loop and J. Blinn, “Real-Time GPU Rendering of Piecewise Algebraic Surfaces,” *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 664–670, 2006.

[LBFH10] F. Latief, B. Biswal, U. Fauzi, and R. Hilfer, “Continuum reconstruction of the pore scale microstructure for fontainebleau sandstone,” *Physica A: Statistical Mechanics and its Applications*, vol. 389, no. 8, pp. 1607–1618, 2010.

[LBH11] N. Lindow, D. Baum, and H.-C. Hege, “Voronoi-Based Extraction and Visualization of Molecular Paths,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 2025–2034, 2011.

[LBM+05] P. Lipowsky, M. J. Bowick, J. H. Meinke, D. R. Nelson, and A. R. Bausch, “Direct visualization of dislocation dynamics in grain-boundary scars,” *Nature Materials*, vol. 4, pp. 407–411, 2005.

[LBM+06] D. Laney, P.-T. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci, “Understanding the Structure of the Turbulent Mixing Layer in Hydrodynamic Instabilities.” *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1053–1060, 2006.

[LBPH10] N. Lindow, D. Baum, S. Prohaska, and H.-C. Hege, “Accelerated Visualization of Dynamic Molecular Surfaces,” *Computer Graphics Forum*, vol. 29, pp. 943–952, 2010.

[LC87] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3D surface construction algorithm,” *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[LCD06] T. Luft, C. Colditz, and O. Deussen, “Image enhancement by unsharp masking the depth buffer,” *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 1206–1213, 2006.

[LD08] A. Lagae and P. Dutré, “A Comparison of Methods for Generating Poisson Disk Distributions,” *Computer Graphics Forum*, vol. 27, no. 1, pp. 114–129, 2008.

[Li03] J. Li, “AtomEye: an efficient atomistic configuration viewer,” *Modelling and Simulation in Materials Science and Engineering*, vol. 11, no. 2, pp. 173–177, 2003.

[LNW+10] H. Li, D. Nehab, L.-Y. Wei, P. V. Sander, and C.-W. Fu, “Fast capacity constrained Voronoi tessellation,” in *Posters Program of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2010, p. 13.

[LR11] F. Lindemann and T. Ropinski, “About the Influence of Illumination Models on Image Comprehension in Direct Volume Rendering,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 1922–1931, 2011.

[LvLRR08] L. Linsen, T. van Long, P. Rosenthal, and S. Rosswog, "Surface Extraction from Multi-field Particle Volume Data Using Multi-dimensional Cluster Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1483–1490, 2008.

[LVRH07] O. D. Lampe, I. Viola, N. Reuter, and H. Hauser, "Two-Level Approach to Efficient Visualization of Protein Dynamics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1616–1623, Nov-Dec 2007.

[Max04] N. Max, "Hierarchical molecular modelling with ellipsoids," *Journal of Molecular Graphics and Modelling*, vol. 23, pp. 233–238, 2004.

[MBW08] O. Mattausch, J. Bittner, and M. Wimmer, "CHC++: Coherent Hierarchical Culling Revisited," *Computer Graphics Forum*, vol. 27, no. 2, pp. 221–230, Apr. 2008.

[MC95] G. Martinelli and M. C. Carotta, "Thick-film gas sensors," *Sensors and Actuators B: Chemical*, vol. 23, no. 2–3, pp. 157–161, 1995.

[McG10] M. McGuire, "Ambient Occlusion Volumes," in *Proceedings of ACM SIGGRAPH/Eurographics High Performance Graphics 2010*, 2010, pp. 47–56.

[Meg] "MegaMol Project Website," http://www.vis.uni-stuttgart.de/megamol/ (last visited: 17.01.2012).

[Mey94] B. Meyer, *Reusable software: the Base Object-Oriented Component Libraries*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[MGE07] C. Müller, S. Grottel, and T. Ertl, "Image-Space GPU Metaballs for Time-Dependent Particle Data Sets," in *Proceedings of VMV*, 2007, pp. 31–40.

[MGK77] J. A. McCammon, B. R. Gelin, and M. Karplus, "Dynamics of folded proteins," *Nature*, vol. 267, no. 5612, pp. 585–590, 1977.

[MHLK05] A. Moll, A. Hildebrandt, H.-P. Lenhof, and O. Kohlbacher, "BALLView: An object-oriented molecular visualization and modeling framework," *Journal of Computer-Aided Molecular Design*, vol. 19, no. 11, pp. 791–800, 2005.

[MI87] S. Murakami and H. Ichihara, "On a 3D Display Method by Metaball Technique," *Electronics Communication Conference*, vol. 70, no. 8, pp. 1607–1615, 1987.

[Mit07] M. Mittring, "Finding Next Gen: Cryengine 2," in *ACM SIGGRAPH 2007 courses*, 2007, pp. 97–121.

[MSD] "Microsoft *DirectX SDK*." http://msdn.microsoft.com/directx/ (last visited: 17.01.2012).

[MTH00] C. Manwart, S. Torquato, and R. Hilfer, "Stochastic Reconstruction of Sandstones," *Physical Review E*, vol. 62, p. 893, 2000.

[Nak04] M. Nakasako, "Water-protein interactions from high-resolution protein crystallography," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 359, no. 1448, pp. 1191–1204, 2004.

[NEA+04] V. Natarajan, P. H. Edelsbrunner, P. L. Arge, P. J. Harer, and P. X. Sun, "Topological Analysis of Scalar Functions for scientific Data Visualization," Ph.D. dissertation, Duke University, 2004.

[NHK+85] H. Nishimura, M. Hirai, T. Kawai, T. Kawata, I. Shirakawa, and K. Omura, "Object Modeling by Distribution Function and a Method of Image Generation," *Electronics Communication Conference*, pp. 718–725, 1985.

[NKV99] A. Nakano, R. K. Kalia, and P. Vashishta, "Scalable Molecular-Dynamics, Visualization, and Data-Management Algorithms for Materials Simulations," *Computing in Science and Engineering*, vol. 1, no. 5, pp. 39–47, 1999.

[NLC02] H.-S. Na, C.-N. Lee, and O. Cheong, "Voronoi diagrams on the sphere," *Computational Geometry*, vol. 23, no. 2, pp. 183–194, 2002.

[NM05] N. Neophytou and K. Mueller, "GPU accelerated image aligned splatting," in *Fourth International Workshop on Volume Graphics*, 2005, pp. 197–242.

[NN94] T. Nishita and E. Nakamae, "A Method for Displaying Metaballs by using Bézier Clipping," *Computer Graphics Forum*, vol. 13, pp. 271–280, 1994.

[Nvi] "Nvidia *Direct3D SDK 10 Code Samples*." http://developer.download.nvidia.com/SDK/10/direct3d/samples.html (last visited: 17.01.2012).

[OHS06] T. Ochotta, S. Hiller, and D. Saupe, "Single-pass high-quality splatting," University of Konstanz, Tech. Rep., 2006, Konstanzer Schriften in Mathematik und Informatik 219.

[OLW91] G. Otting, E. Liepinsh, and K. Wüthrich, "Protein hydration in aqueous solution," *Science*, vol. 254, no. 5034, pp. 974–980, Nov 1991.

[Opea] OpenGL Extension Registry, "NV_conditional_render," http://www.opengl.org/registry/specs/NV/conditional_render.txt (last visited: 17.01.2012).

[Opeb] "OpenGL3.0 BOF presentations at SIGGRAPH 2007," http://www.khronos.org/library/detail/siggraph_2007_opengl_birds_of_a_feather_bof_presentation/.

[Ope08] OpenMP Architecture Review Board, "OpenMP Application Program Interface," Specification, 2008. [Online]. Available: http://-www.openmp.org/mp-documents/spec30.pdf (last visited: 17.01.2012)

[Paj03] R. Pajarola, "Efficient Level-of-Details for Point Based Rendering," in *Proceedings of IASTED International Conference on Computer Graphics and Imaging (CGIM)*, 2003, pp. 141–146.

[Par72] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[PCG+92] G. Perrot, B. Cheng, K. D. Gibson, J. Vila, K. A. Palmer, A. Nayeem, B. Maigret, and H. A. Scheraga, "MSEED: A Program for the Rapid Analyt-

ical Determination of Accessible Surface Areas and Their Derivatives," *Journal of Computational Chemistry*, vol. 13, no. 1, pp. 1–11, 1992.

[PF05] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[PG04] M. Pharr and S. Green, "Ambient Occlusion," in *GPU Gems*, R. Fernando, Ed. Addison-Wesley, 2004, pp. 667–692.

[PGH+04] E. F. Pettersen, T. D. Goddard, C. C. Huang, G. S. Couch, D. M. Greenblatt, E. C. Meng, and T. E. Ferrin, "UCSF Chimera–a visualization system for exploratory research and analysis," *Journal of Computational Chemistry*, vol. 25, no. 13, pp. 1605–1612, Oct 2004, http://www.cgl.ucsf.edu/chimera/ (last visited 17.06.2012).

[Phi98] D. Phillips, *The Software Project Manager's Handbook: Principles that Work at Work*. IEEE Computer Society, 1998.

[PJ88] M. Page-Jones, *The practical guide to structured systems design: 2nd edition*. Yourdon Press, 1988.

[PMP10] G. Papaioannou, M. L. Menexi, and C. Papadopoulos, "Real-Time Volume-Based Ambient Occlusion," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, pp. 752–762, September 2010.

[PS06] N. Prabhu and K. Sharp, "Protein-solvent interactions," *Chemical Reviews*, vol. 106, pp. 1616–1623, 2006.

[PSG04] R. Pajarola, M. Sainz, and P. Guidotti, "Confetti: Object-Space Point Blending and Splatting," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 5, 2004, pp. 598–608.

[RCK09] J. Ryu, Y. Cho, and D.-S. Kim, "Triangulation of molecular surfaces," *Computer-Aided Design*, vol. 41, pp. 463–478, June 2009.

[RE05] G. Reina and T. Ertl, "Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization," in *Computer Graphics Forum*, 2005, pp. 177–182.

[Rei08] G. Reina, "Visualization of Uncorrelated Point Data," Dissertation, Visualisierungsinstitut der Universität Stuttgart, Stuttgart, Deutschland, 2008.

[Rel96] Reliability Analysis Center, *Introduction to Software Reliability: A state of the Art Review*. Reliability Analysis Center (RAC), 1996.

[Ric77] F. M. Richards, "Areas, Volumes, Packing, and Protein Structure," *Annual Review of Biophysics and Bioengineering*, vol. 6, no. 1, pp. 151–176, 1977.

[RL00] S. Rusinkiewicz and M. Levoy, "QSplat: A Multiresolution Point Rendering System for Large Meshes," in *ACM SIGGRAPH*, 2000, pp. 343–352.

[RPK07] J. Ryu, R. Park, and D.-S. Kim, "Molecular surfaces on proteins via beta shapes," *Computer-Aided Design*, vol. 39, no. 12, pp. 1042–1057, 2007.

[RPZ02] L. Ren, H. Pfister, and M. Zwicker, "Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering." in *Computer Graphics Forum*, vol. 21, no. 3, 2002, pp. 461–470.

[SA07] P. Shanmugam and O. Arikan, "Hardware accelerated ambient occlusion techniques on GPUs," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 2007, pp. 73–80.

[SA10] A. Stukowski and K. Albe, "Extracting dislocations and non-dislocation crystal defects from atomistic simulation data," *Modelling and Simulation in Materials Science and Engineering*, vol. 18, no. 8, pp. 85001–85013, 2010.

[San04] B. Sanjeev, "Ankush," 2004, http://www.geocities.ws/dviip/ (last visited: 17.01.2012).

[SBSO09] J. Shopf, J. Barczak, T. Scheuermann, and C. Oat, "Deferred Occlusion from Analytic Surfaces," in *Shader X7: Advanced Rendering Techniques*. Charles River Media, 2009, pp. 445–454.

[SCWS04] P. Schall, I. Cohen, D. A. Weitz, and F. Spaepen, "Visualization of Dislocation Dynamics in Colloidal Crystals," *Science*, vol. 24, pp. 1944–1948, 2004.

[SCWS06] ——, "Visualizing dislocation nucleation by indenting colloidal crystals," *Nature*, vol. 440, pp. 319–323, 2006.

[SEBH02] J. Schmidt-Ehrenberg, D. Baum, and H. C. Hege, "Visualizing dynamic molecular conformations," in *Proceedings of IEEE Visualization 2002*, 2002, pp. 235–242.

[Set06] J. P. Sethna, *Statistical Mechanics: Entropy, Order Parameters and Complexity*. Oxford University Press: Oxford, New York, 2006.

[SGEK97] T. C. Sprenger, M. H. Gross, A. Eggenberger, and M. Kaufmann, "A Framework for Physically-Based Information Visualization," in *Proceedings of Eurographics Workshop on Visualization 1997*, 1997, pp. 77–86.

[SGS06] C. Stoll, S. Gumhold, and H.-P. Seidel, "Incremental Raycasting of Piecewise Quadratic Surfaces on the GPU," in *IEEE Symposium on Interactive Ray Tracing 2006*, 2006, pp. 141–150.

[SHG09] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–10.

[SML97] W. J. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object Orient Approach to Computer Graphics, Second Edition*, ser. Kitware Inc. Prentice Hall, 01 1997.

[SN07] J. M. Singh and P. J. Narayanan, "Real-Time Ray-Tracing of Implicit Surfaces on the GPU," International Institute of Information Technology, Hyderabad, India, Tech. Rep., Jul 2007.

[SN09] J. M. Singh and P. Narayanan, "Real-Time Ray Tracing of Implicit Surfaces on the GPU," *IEEE Transactions on Visualization and Computer Graphics*, vol. 99, pp. 261–272, 2009.

[SNK+03] A. Sharma, A. Nakano, R. K. Kalia, P. Vashishta, S. Kodiyalam, P. Miller, W. Zhao, X. Liu, T. J. Campbell, and A. Haas, "Immersive and interactive

exploration of billion-atom systems," *Presence*, vol. 12, no. 1, pp. 85–95, 2003.

[SO97] M. F. Sanner and A. J. Olson, "Real time surface reconstruction for moving molecular fragments," in *Pacific Symposium on Biocomputing*, 1997, pp. 385–396.

[SOS96] M. F. Sanner, A. J. Olson, and J.-C. Spehner, "Reduced Surface: An Efficient Way to Compute Molecular Surfaces," *Biopolymers*, vol. 38, no. 3, pp. 305–320, Dec 1996.

[SPL04] M. Sainz, R. Pajarola, and R. Lario, "Points Reloaded: Point-Based Rendering Revisited ," in *Proceedings of Symposium on Point-Based Graphics 2004*, 2004, pp. 121–128.

[SSKE05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl, "A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting," in *International Workshop on Volume Graphics 2005*, 2005, pp. 187–195.

[SSZK04] M. Sattler, R. Sarlette, G. Zachmann, and R. Klein, "Hardware-accelerated ambient occlusion computation," in *Proceedings of VMV*, 2004, pp. 331–338.

[Sto98] J. Stone, "An Efficient Library for Parallel Ray Tracing and Animation," Master's thesis, University of Missouri-Rolla, Department of Computer Science, 1998.

[Sup83] K. J. Supowit, "The Relative Neighborhood Graph, with an Application to Minimum Spanning Trees," *Journal of the ACM*, vol. 30, no. 3, pp. 428–448, 1983.

[SVH06] T. Schnabel, J. Vrabec, and H. Hasse, "Molecular Modeling of Hydrogen Bonding Fluids: Monomethyamine, Dimethylamine, and Water Revised," in *High Performance Computing in Science and Engineering*, W. E. Nagel, W. Jäger, and M. Resch, Eds. Springer, 2006, pp. 515–525.

[SWBG06] C. Sigg, T. Weyrich, M. Botsch, and M. H. Gross, "GPU-Based Ray-Casting of Quadratic Surfaces," in *Proceedings of Symposium on Point-Based Graphics 2006*. Eurographics Association, 2006, pp. 59–65.

[TA95] M. Totrov and R. Abagyan, "The Contour-Buildup Algorithm to Calculate the Analytical Molecular Surface," *Journal of Structural Biology*, vol. 116, pp. 138–143, 1995.

[TCM06] M. Tarini, P. Cignoni, and C. Montani, "Ambient Occlusion and Edge Cueing for Enhancing Real Time Molecular Visualization." *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1237–1244, 2006.

[TGN+06] E. Tejada, J. Gois, L. G. Nonato, A. Castelo, and T. Ertl, "Hardware-accelerated Extraction and Rendering of Point Set Surfaces," in *Proceedings of Eurographics/IEEE VGTC Symposium on Visualization*, 2006, pp. 21–28.

[THM+03] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross, "Optimized Spatial Hashing for Collision Detection of Deformable Objects," in *Proceedings of VMV*, 2003, pp. 47–54.

[TL04] R. Toledo and B. Lévy, "Extending the graphic pipeline with new gpu-accelerated primitives," INRIA Lorraine, Tech report, 2004.

[TL08] R. Toledo and B. Levy, "Visualization of industrial structures with implicit gpu primitives," in *International Symposium on Visual Computing*, 2008, pp. 139–150.

[TvW99] A. Telea and J. J. van Wijk, "Simplified representation of vector fields," in *Proceedings IEEE Visualization 1999*, 1999, pp. 35–42.

[VBW94] A. Varshney, F. P. Brooks, and W. V. Wright, "Linearly Scalable Computation of Smooth Molecular Surfaces," *IEEE Computer Graphics and Applications*, vol. 14, no. 5, pp. 19–25, 1994.

[VHS01] J. M. Verner and B. Henderson-Sellers, "A Pilot Study in Effort Estimation for the Generalization of Object-Oriented Components for Reuse," in *Proceedings of the 13th Australian Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 190–199.

[vKvdBT07] K. van Kooten, G. van den Bergen, and A. Telea, "Point-Based Visualization of Metaballs on a GPU," in *GPU GEMS 3*, H. Nguyen, Ed. Addison-Wesley, 2007.

[vMAF+07] J. A. van Meel, A. Arnold, D. Frenkel, S. F. P. Zwart, and R. G. Belleman, "Harvesting graphics power for MD simulations," *Molecular Simulation*, vol. 34, no. 3, pp. 259–266, 2007.

[VSH01] J. Vrabec, J. Stoll, and H. Hasse, "A Set of Molecular Models for Symmetric Quadrupolar Fluids," *Journal of Physical Chemistry B*, vol. 105, pp. 12126–12133, 2001.

[VW26] M. Volmer and A. Weber, "Keimbildung in gesättigten Gebilden," *Zeitschrift für Physikalische Chemie A*, vol. 119, pp. 277–301, 1926.

[vW05] J. van Wijk, "The Value of Visualization," in *Proceedings of IEEE Visualization*, 2005, pp. 79–86.

[WE98] R. Westermann and T. Ertl, "Efficiently using graphics hardware in volume rendering applications," in *Proceedings of SIGGRAPH 1998, the 25th annual Conference on Computer Graphics and Interactive Techniques*, 1998, pp. 169–177.

[Wea04] C. Weaver, "Building Highly-Coordinated Visualizations in Improvise," *IEEE Symposium on Information Visualization*, pp. 159–166, 2004.

[Wes90] L. Westover, "Footprint evaluation for volume rendering," *ACM SIGGRAPH 1990 Computer Graphics*, vol. 24, no. 4, pp. 367–376, 1990.

[Wil06] G. Wilson, "Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive," *Computing in Science and Engineering*, vol. 8, no. 6, pp. 66–69, Nov.-Dec. 2006.

[WMW86] G. Wyvill, C. McPheeters, and B. Wyvill, "Data structure for soft objects," *The Visual Computer*, vol. 2, no. 4, pp. 227–234, 1986.

[WST+07] T. Weinkauf, J. Sahner, H. Theisel, H.-C. Hege, and H.-P. Seidel, "A unified feature extraction architecture," in *Active Flow Control*, ser. Notes on Numerical Fluid Mechanics and Multidisciplinary Design (NNFM), R. King, Ed., 2007, pp. 119 – 133.

[ZIK98] S. Zhukov, A. Iones, and G. Kronin, "An Ambient Light Illumination Model," in *Proceedings of Eurographics Rendering Workshop 1998*, 1998, pp. 45–56.

[ZPvBG01] M. Zwicker, H. Pfister, J. van Baar, and M. Gross, "Surface splatting," in *ACM SIGGRAPH 2001*, 2001, pp. 371–378.

[ZSP08] Y. Zhang, B. Solenthaler, and R. Pajarola, "Adaptive Sampling and Rendering of Fluids on the GPU," in *Eurographics/IEEE VGTC Symposium on Volume and Point-Based Graphics*, 2008, pp. 137–146.

[ZXB07] W. Zhao, G. Xu, and C. Bajaj, "An algebraic spline model of molecular surfaces," in *Proceedings of the ACM symposium on Solid and physical modeling*, 2007, pp. 297–302.