Visualization of Uncorrelated Point Data

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

Vorgelegt von

Guido Reina

aus Ostfildern-Ruit

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. H.-J. Bungartz
Tag der mündlichen Prüfung:	16. 09. 2008

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2008

Acknowledgements

During the making of this dissertation I have been helped and supported by a number of people, for things big and small, related and unrelated to the dissertation itself. So if anything, it has helped reduce the many delays the final thesis has suffered. First of all my thanks go to Thomas Ertl for advising me and of course for having the patience to bear with the aforementioned factors. I am also grateful to the second referee Hans-Joachim Bungartz for encouragement and the painstaking review of this work. Thanks are owed to the DFG (the German Research Foundation) for funding as part of SPP 1041 "Verteilte Vermittlung und Verarbeitung Digitaler Dokumente" (Distributed Processing and Delivery of Digital Documents) and the SFB 716 "Dynamische Simulation von Systemen mit großen Teilchenzahlen" (Dynamic simulation of systems with large numbers of particles). I also want to thank the Landesstiftung Baden-Württemberg for funding in the context of project 688 "Massiv parallele molekulare Simulation und Visualisierung der Keimbildung in Mischungen für skalenübergreifende Modelle".

Since there is much to acknowledge, thanks are in no particular order, since I probably could not come up with one even if forced.

Regarding the work that is not part of the thesis, I want to thank Dirc Rose, Simon Stegmaier and Daniel Weiskopf for giving me the opportunity to work with them on a novel way of automatically bringing menu-based interaction onto hand-held devices [RSR+03]. I want to thank my diploma thesis advisors, Sven Lange-Last and Klaus Engel, also for helping me transforming the resulting graph-oriented mainframe device management prototype into a paper afterwards [RLLEE03]. Katrin Bidmon and Fabian Bös worked with great enthusiasm on our time-based haptic interface for proteins [BRB+07]. It has been a real pleasure working with all those people. I also enjoyed to digress a bit from the main focus of this thesis every now and then and it certainly helped putting things into perspective.

I want to thank Matthias Hopf for infecting me with the point-cloud-rendering virus, lots of discussion about his point splatting, and finally for handing down his framework for further refinement/abuse by molecular glyphs, which would in the end set the main focus of this thesis. I guess if anyone is to blame for the overall outcome of my research, it might be him. I also want to thank Dirc Rose for some fruitful discussion about the first prototype of GPU-based dipoles, which sometimes did not really want to work the way I wanted them

to.

Many thanks go to Katrin Bidmon and Sebastian Grottel, who both worked closely with me on two of the central aspects of this thesis: extremely complex raycast glyphs and time-based visualization as well as a complete overhaul of the point-rendering framework. Katrin worked a lot with me for a representation of elliptic-profile tubes that finally could be handled on the GPU and she did not falter once even though it always cost us more time than even we had anticipated. Frank Enders provided us with the datasets we could test our approach with as well as the needed expertise about the data and also gave us access to the MedAlyVis[1] code as to save precious time by integrating our approach directly into this application. Sebastian Grottel spent much of his time programming for or with me starting from his diploma thesis aimed at extending the original code from Matthias to a complete and thorough re-design of the whole framework. Too much discussion took place about so many aspects of the framework (also used for benchmarking the critical aspects as described later in this thesis) and the cluster visualization and analysis to be detailed here.

Jadran Vrabec and Martin Horsch proved to be excellent colleagues as well as Martin Bernreuther. The simulation code and results that most of the molecular visualization builds upon were provided by them along with repeated discussion which in the end pushed things forward considerably for all of us. Thank you also for so much initiative, which is a rare and precious thing for interdisciplinary collaboration.

I really passed quality time with my several room mates over the years; I think I might even be holding the record for most office switches in the whole institute. After some initial loneliness I joined forces with Manfred Weiler, first in the smelly office in the old building, then in the aquarium in the new building, always amused by the things you better had not done to OpenSG and volume rendering in general. After that, I was exiled to the far end of the building together with Joachim Diepstraten and Marcelo Magallon. Still wondering whether Heihachi Mishima might be dead, I moved a couple of metres into Katrin Bidmon's and Dirc Rose's office, where I found out what kind of things you really wanted to do with our building and a polygonal model and on-site photographs of it. The Bidmon/Reina configuration even survived the move to the VISUS premises as well as the first re-arrangements therein a year after. I'm not sure whether one of these combinations was particularly productive, however I know I had fun. Which might have been a good reason to move this chapter to the opposing end of the thesis, just in case.

I'm sure though that with Katrin I had it a lot easier with my constant struggles with LaTeX, since of course I always wanted to get the text out a bit differently

[1] a medical visualization framework developed at the Neurozentrum Erlangen-Nürnberg in the context of project C9 in the SFB 603

from the way LaTeX supposed would be good for me.

Fruitful discussions as well as much help that also went into the many small things that sometimes group up and try to keep you from doing the work you really needed to get done yesterday were offered, in no particular order, by Mike Eissele, Thomas Klein, Magnus Strengert, and especially Christoph Müller, who will be still working with me for a while on the huge high-res display project for VISUS.

Special thanks go to my grandfather and grandmother (may she rest in peace), who offered their unconditional support throughout all of my studies. And of course to my friends outside the VIS institute who helped distracting me from work when I could really use it: Oliver Hasprich, Florian Meister and Peter Oberparleiter. Last but not least I thank Marion Freese for a wonderful time.

A special mention goes to those who proof-read (parts) of my thesis: Marion Freese, Florian Meister, Katrin Bidmon, Sebastian Grottel, Steffen Koch, and Magnus Strengert.

I'm quite sure I forgot many people, so I want thank those for all their help and support and apologize while trying to blame it on the non-existing structure of this section.

Abstract and Chapter Summaries

Abstract

Sciences are the most common application context for computer-generated visualization. Researchers in these areas have to work with large datasets of many different types, but the one trait that is common to all is that in their raw form they exceed the cognitive abilities of human beings. Visualization not only aims at enabling users to quickly extract as much information as possible from datasets, but also at allowing the user to work at all with those that are too large and complex to be directly grasped by human cognition. In this work, the focus is on uncorrelated point data, or point clouds, which is sampled from real-world measurements or generated by computer simulations. Such datasets are gridless and exhibit no connectivity, and each point represents an entity of its own. To effectively work with such datasets, two main problems must be solved: on the one hand, a large number of complex primitives with potentially many attributes must be visualized, and on the other hand the interaction with the datasets must be designed in an intuitive way.

This dissertation will present novel methods which allow the handling of large, point-based data sets of high dimensionality. The contribution for the rendering of hundreds of thousands of application-specific glyphs is a *Graphics-Processing-Unit*(GPU)-based solution that allows the exploration of datasets that exhibit a moderate number of dimensions, but an extremely large number of points. These approaches are proven to be working for *molecular dynamics*(MD) datasets as well as for 3D tensor fields. Factors critical for the performance of these algorithms are thoroughly analyzed, the main focus being on the fast rendering of these complex glyphs in high quality. To improve the visualization of datasets with many attributes and only a moderate number of points, methods for the interactive reduction of dimensionality and analysis of the influences of different dimensions as well as of different metrics will be presented. The rendering of the resulting data in 3D similarity space is also addressed. A GPU-based reduction of dimensions has been implemented that allows interactive tweaking of the reduction parameters while observing the results in real time.

With the availability of a fast and responsive visualization, the missing component for a complete system is the human-computer interaction. The user must be able to navigate the information space and interact with a dataset,

selecting or filtering the items that are of interest to him, inspecting the attributes of particular data points. Today, one must distinguish between the application context and the modality of different interaction approaches. Current research ranges from keyboard-and-mouse desktop interaction over different haptic interfaces (also including feedback) up to tracked interaction for *virtual reality*(VR) installations.

In the context of this work, the problem of interacting with point-based datasets is tackled for two different situations. The first is the workstation-based analysis of clustering mechanics in thermodynamics simulations, the second a VR immersive navigation and interaction with point cloud datasets.

Chapter Summaries

Chapter 1

The first chapter introduces the scenario for this dissertation, which mainly focuses on large, point-based datasets with several attributes per point. The motivation for especially dealing with three major aspects (GPU usage, dimensionality reduction, interaction) is given and the project context as well as the cooperations that took place during the evolution of this thesis are explained.

Chapter 2

The visualization pipeline is explained with its different stages, as its structure is reflected in the structure of this dissertation as a whole. The various data types that make up the base for the different visualization algorithms are introduced and also put into context with respect to the following chapters. Modern graphics hardware is explained along with its capabilities and programmable features. The basics of volume rendering are summarized as well as the background for human-computer interaction.

Chapter 3

This chapter focuses on the efficient rendering of large datasets with the help of GPU-generated glyphs based on implicit geometric surfaces. The employed datasets, generated from molecular dynamics simulations, are described along with common visualization approaches and previous work for GPU-based glyphs. Initially, a thorough analysis of the available options for representing data and efficiently uploading it to the graphics card is given, then several simple and compound glyphs are presented along with details for the optimized rendering. An application for the time-based visualization of MD datasets that builds upon the presented techniques is introduced along with abstract visu-

alizations that facilitate the special case of nucleation simulation research and more specifically the cluster detection and its evolutionary tracking. In the following, the more complex field of diffusion tensor visualization is explained in the context of a medical application. A very sophisticated glyph for the representation of the whole eigensystem of such tensors is introduced and its efficient implementation is detailed. Results from a real-world dataset are employed to emphasize the appropriateness of the proposed approach.

Chapter 4

Since some of the data worked on during the dissertation had a number of attributes that exceeded the possibilities for sensibly representing them all, an approach for interactively reducing the attributes to a similarity measure and then investigating the dataset structure to uncover clusters and trends is presented. Volume rendering is applied to the reduced data since in this way a constant minimum performance can be offered regardless of the dataset size. An example application with a publicly available high-dimensional dataset is described, also to underline the potential of making visual queries through multiple transfer functions that can be applied to the volume representation. This approach relies on incremental refinement by the user and because of this iterative workflow it requires the best possible performance that is available for dimensionality reduction, which in turn led to a GPU-assisted implementation of the employed algorithm (FastMap). This last part also contains some details on performance implications of texture management, which emerged during the optimization process.

Chapter 5

This chapter focuses on the filtering stage of the visualization pipeline, more specifically the interactive filtering via several selection mechanisms on the one hand on normal workstations and on the other hand on VR displays with tracked interaction. Special care is taken of the implications of this last mode, for example the caused fatigue, input precision, and also usability for the more complex interactions. A novel metaphor is introduced as well, which can also be employed for improving the performance of the rendering or show more details in case of an adaptive algorithm like in the presented one. The usability is tested against the principles brought forth by Norman [Nor88] and is also subjected to a user study, and thus proven to be effective and efficient.

Chapter 6

The last chapter points out the contributions of this thesis and explains the process that underlies the development of the particular, but complete implemen-

tation of the visualization pipeline that this thesis represents from a software engineering point of view. It points out the factors that need be considered when developing such a solution and how they interact. The rest of the thesis is used for examples. A short outlook on possible future developments is also given.

Contents

1 Introduction

Sciences are among the most common application contexts for computer-generated visualization. Researchers in these areas have to work with large datasets of many different types, but the one trait that is common to all is that in their raw form they exceed the cognitive abilities of human beings. Visualization not only aims at enabling users to quickly extract as much information as possible from datasets, but also at allowing the user to work with those that are too large and complex to be directly grasped by human cognition. In addition to the task of developing metaphors to present certain types of datasets in the first place, the dataset sizes themselves are becoming more and more of a problem in terms of processing power and storage requirements. Technological progress on the one hand allows to capture ever larger datasets of varying dimensionality, in part even with few to no interaction from a human operator, thus setting virtually no limit to the frequency or conditions of capture. On the other hand computer-based simulation has been widely adopted for the investigation of systems with problematic parameters – either dangerous or under extreme conditions and as such very costly – saving money and cutting down on the risk for incidents. The size of thusly generated datasets is only limited by the available processing power and storage. As much as the research is eased by the availability of large numbers of samples, as growingly realistic it has become to compare simulation results with real-life experiments, the much harder it is to interpret these vast amounts of data. This problem partly stems from the habit of many researchers to inspect the resulting data in raw, tabular form and thus suffer from the many problems that come with this kind of output format: the sheer size makes it nearly impossible to compare different parts of the dataset because they are spread too far apart, and the comparison also requires a higher cognitive effort – it is far easier and quicker to compare the lengths of multiple bars in a bar-chart than several floating-point figures, to just make an example. Consequently, many research areas can benefit already from the availability of a visualization, however the step from improving a problematic situation to an efficiently manageable one (from the view of the human cognitive system) usually has to be much larger.

1.1. Uncorrelated Point Data

Uncorrelated point data in the context of this work denotes datasets that consist of items that have several (and potentially a great many) attributes, among which may be an explicit position in $\mathbb{R}^n$, and that are regarded as individual entities, i. e., they are not part of a surface/mesh and thus do not exhibit connectivity information. Subsampling and interpolation are not straightforward and do not only induce information loss, but might also generate false data (cfr. the interpolation of categorical values). No requirements exist as for the dependency of the different attributes among each other. The individuality of the items is the reason why the starting point for visualization in this work is always a direct approach.

1.2. Motivation

To obtain an effective and efficient visualization of such datasets, several problems must be solved. Taking a look at the visualization pipeline (see figure 2.1), there are different stages that can be influenced. In this work the main focus is on the rendering stage, introducing new algorithms that take advantage of modern programmable Graphics Processing Units (see chapter 3). However this stage can only be capitalized on when appropriate data is available. If this is not the case and the data is too complex to be directly visualized, one option is the introduction of dimensionality reduction algorithms into the mapping step, which will be elaborated upon in chapter 4. For maximum effectiveness, the user of the resulting system needs one more component to have control over the performed operations, which is the user interface, the last aspect that will be considered to obtain a fully functional solution (see chapter 5). The user interface also allows for control over the available filtering mechanisms, resulting in one coherent implementation of the visualization pipeline. The related problems of navigation and data inspection are also looked at in this work.

1.2.1. GPU-accelerated Rendering

To render large datasets with a certain number of attributes, using glyphs is a common approach nowadays. The basic idea consists of developing an arbitrary geometry that can convey multiple values via its features and render one of these glyphs per data point. The shape of these glyphs directly depends on the dataset, i.e. its dimensionality, and application area where the visualization will be employed [Che73, PG88].

The datasets available in the context of this work usually consist of several tens to hundreds of thousands data points, which makes the rendering method a

crucial choice for the resulting performance. Besides choosing an algorithm that performs well for the high number of primitives needed, it has also proven beneficial for inherently parallelizable algorithms to be executed at least in part on the Graphics Processing Unit. Currently, GPUs are being developed at increasingly fast rates and have long since passed the complexity and parallelism of CPUs (see table 1.1). The same holds for the GPU memory bandwidth, but in terms of flexibility CPUs still have a significant advantage (see also chapter 2). Admittedly the driving force behind the fast development of GPUs is the computer gaming industry, however the demand for more realism, that is, more complex scenes and better special effects, coincides with the requirements for real time visualization algorithms, which greatly benefit from the processing power of GPUs if the algorithms can be adapted to make use of the *Single Instruction, Multiple Data* (SIMD) architecture of graphics cards.

Processor	Transistors	Parallelism	Memory B/W
Intel Pentium 4[1]	169M	4	6.4GB/s
AMD Opteron 275	233M	4	22.4GB/s
AMD Athlon FX-62	227M	4	12.8 GB/s
Intel Core 2 Duo[2]	291M	4	8.5 GB/s
Nvidia GeForce 7900 GTX	302M	24	51.2GB/s
Nvidia GeForce 8800 Ultra	681M	128	103.7GB/s
Nvidia GeForce GTX 280	1,400M	240	141.7GB/s

Table 1.1.: Some technical specs of current CPUs and GPUs. Parallelism denotes the number of single-precision float operations that can be executed in parallel. From the Nvidia GeForce 8800 Ultra onwards it is the theoretical maximum because of the unified shader architecture.

[1] Prescott; 2MB Cache [2] 1066MHz FSB

One problem with GPUs is, however, the large, but still quite limited bandwidth available for uploading data from the CPU to the GPU. Since graphics hardware basically works only with triangles, the standard approach for rendering complex glyphs is to tessellate them into meshes, thus obtaining (potentially a great many) triangles, usually in proportion to the glyph complexity. This legacy approach was the only option when GPUs only had a fixed-function pipeline, and the geometry generation had to be accomplished exclusively on the CPU. If the dataset is not static, no display lists or vertex buffers can be used and all of these triangles have to be transferred to the GPU once per rendered frame, consuming a significant amount of bandwidth on the graphics bus as well as space and bandwidth of the system memory. Static datasets could be kept in the graphics card memory, which ensures much better rendering performance, but consequently limits the data set size.

Because of the flexible programmability and the high parallelism in current GPUs, much more can be computed on the fly, so a viable alternative is to render the glyphs from some implicit representation on the GPU and only send parameters over the system bus to conserve bandwidth. The proposed solution is inspired by the raycasting approach of Gumhold [Gum03], which uploads just a billboard geometry to the graphics card and uses the programmable fragment shaders of GPUs to intersect an accordingly parameterized implicit ellipsoid surface on a per-fragment basis. This idea will be extended to much more complex glyphs in chapter 3, generally trading high bandwidth requirements for a high fragment processing load, but yielding high-quality glyphs that can be rendered in real time despite their large numbers.

1.2.2. Dimensionality Reduction

Large datasets do not necessarily contain many points. There are also fields where only thousands of samples are collected, which however can have hundreds of attributes each. Census data or cancer screening datasets in chemistry are representatives of this category. Rendering that many attributes with glyphs is practically impossible since all those features require too much space and thus restrict the number of glyphs that can be arranged on the screen simultaneously too much to be useful (see also chapter 4).

A viable solution that is also widely used in the information visualization community is the abstraction from the concrete attributes by mapping the objects into a low-dimensional space $\mathbb{R}^m, 1 \leq m \leq 3$ such that the intrinsic structure of the data is preserved. Usually this structure should be dictated by the inter-object proximity – or similarity – such that the high-dimensional relationships are depicted as accurately as possible by low-dimensional distances. *Multi-dimensional scaling* (MDS) is one of the reference methods to achieve such a mapping [BG97]. The basic concept is an optimization problem to minimize the difference between the high-dimensional and the low-dimensional coordinates by using any one from a variety of algorithms.

In this work, two main contributions to improve the handling of such high-dimensional datasets are based on the alternative algorithm for dimensionality reduction called *FastMap* [FL95] which can mainly offer a much better performance than MDS. Chapter 4 will present an application that allows the user to flexibly parameterize the calculations to investigate how the different attributes affect the results and whether non-metric distance measures can help improve the structure a dataset exhibits. The resulting data is rendered as a scatterplot to allow for the visual detection of cluster structures which are indicators for common properties in data points. Implementing this algorithm in graphics hardware is shown to offer real-time interactivity, which makes the user-driven optimization process more efficient than with CPU-based implementations.

1.2.3. Interaction

However good a visualization may be, the user can only start to take advantage of it the moment he can influence which part of the data he is observing and which attributes he is interested in. This parametrization of the visualization pipeline must be made available with an interface that allows the user to quickly define his exact requirements for all of the stages: filter out data he does not need, tweak the mapping to emphasize the critical attributes, and last, but not least, navigate the rendered data to generate some insight. A visualization that facilitates such a kind of workflow is considered a successful one.

Display, keyboard, and mouse are the prevalent interface most users know day-to-day use of computers, however I wanted to shift the focus to the particular class of stereoscopic output devices that are employed to generate *virtual reality* (VR) environments, which give the user the impression of participating in the space where visualized data is rendered. With the availability of immersive displays like CAVEs, powerwalls and autostereoscopic displays, the need for matching interaction devices and metaphors that do not disrupt such an experience – or even better, emphasize it – is stronger than ever. There have been several efforts to adapt existing devices, like PDAs and laser pointers, to improve the interaction with VR, as well as developing new devices and concepts, for example to deal with displays much larger than the workplace norm [BSW06]. Another popular alternative is the use of tracking systems with or without markers to capture user movement or gestures without disrupting the immersiveness of the VR output.

A novel interaction approach for VR based on optical tracking in conjunction with a stereo-enabled projection is presented in chapter 5. It supports intuitive manipulation of the dataset as well as different metaphors for the selection and filtering in extremely large datasets.

1.3. Context and Cooperations

All of the work for this dissertation has been performed at the Visualization and Interactive Systems Group (VIS) of the Universität Stuttgart. I have worked on different projects, hence this work is partly financed by the DFG in the context of the SPP 1041 $V^{III}D^{II}$ "Distributed Processing and Dissimination of Digital Documents" and partly by the Landesstiftung Baden-Württemberg in the context of Project 688 "Massiv parallele molekulare Simulation und Visualisierung der Keimbildung in Mischungen für skalenübergreifende Modelle". The latter work has been continued, and is still ongoing in the SFB 716 "Dynamic simulation of systems with large numbers of particles", financed again by the DFG.

The work about dimensionality reduction was mainly realized for $V^{III}D^{II}$ in cooperation with Frank Oellien and W. D. Ihlenfeldt from the TORVS Group of the University of Erlangen-Nuremberg. The other cooperation with the University of Erlangen-Nuremberg was thanks to Frank Enders from the Neurocenter/Computer Graphics Group with the aim of realizing GPU-based streamtubes.

The molecular visualization was implemented for project 688 in close collaboration with Jadran Vrabec and Martin Horsch from the Institute of Thermodynamics and Thermal Process Engineering (ITT) in Stuttgart, while we also worked on protein visualization and haptics [BRB+07] with Fabian Bös from the Institute of Technical Biochemistry (ITB). Both put several interesting datasets at our disposal with which to test our applications and which can be seen in most of the molecular dynamics simulation screenshots in this thesis.

Several projects about simulation steering took place in collaboration with Martin Bernreuther, first working at the Simulation of Large Systems group (SgS), and later at the High Performance Computing Center Stuttgart (HLRS). During this project we also collaborated with Andrea Wix from the Institut für Technische Thermodynamik und Kältetechnik of the University of Karlsruhe.

While I was working at VIS, I worked with many colleagues on the different ideas which were published. In chronological order they are: Dirc Rose, Simon Stegmaier and Daniel Weiskopf for [RSR+03], Klaus Engel and Sven Lange-Last for [RLLEE03], Alex Rosiuta for his diploma thesis and consequently [RRE06], and Katrin Bidmon for [BRB+07]. Katrin Bidmon also worked substantially for the hyperstreamlines that are presented in Section 3.3.3 [RBE+06]. I collaborated with Thomas Klein for our chapter in [RKE07]. Sebastian Grottel designed the time-based functionality for point rendering [GRVE07] in his diploma thesis as well as rewriting/redesigning the framework with the help of which all updated comparative performance measurements in chapter 3 have been conducted and which resulted in [GRE09].

2 Fundamentals

This document borrows its basic structure from the different stages of the visualization pipeline, which as such will be described in more detail in the following. Additionally, fundamental concepts about the data and data structures as well as the hardware-accelerated generation of visual output for end users will be introduced, to be built upon in the later chapters of this dissertation.

2.1. The Visualization Pipeline

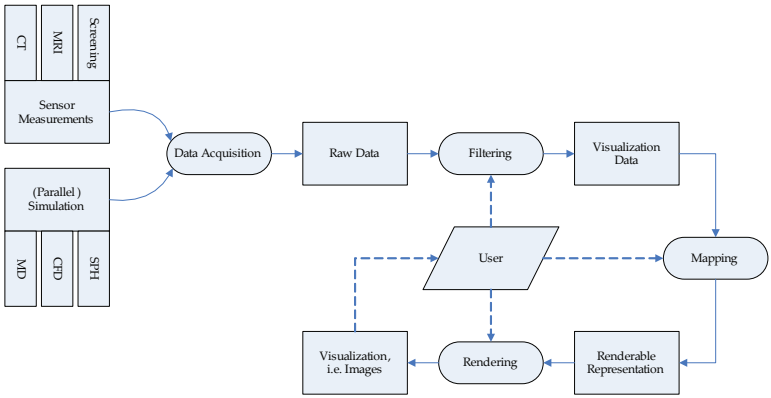


Figure 2.1.: The visualization pipeline.

The task of scientific visualization is to enable a user to discover, understand and solve scientific problems by transforming data about the physical world via mathematical methods and computer-based image generation into something that can easily be interpreted by the human mind (and thus triggering some insight [REE+94]). The process that transforms the input (raw) data into

images is called the *visualization pipeline* and consists of several stages which can all be influenced by the user, thus resulting in a feedback loop if the user is added to the flow as well (see figure 2.1). The first stage, filtering, allows for the selection of specific subsets of (often too large and complex) data and is usually driven by the user's interest in specific aspects. The mapping stage transforms the remaining data into a directly renderable representation, be it geometry, a point cloud, or samples on a grid. This data will usually consist of implicit or explicit positions in $\mathbb{R}^n, 1 \leq n \leq 3$ and attributes like color, opacity, a texture (coordinate) etc. This representation will be rendered in the last stage, usually taking advantage of special hardware and/or graphics processing units. The feedback from the user requires a minimal responsiveness of the system implementing the visualization pipeline (defined at about 10Hz), enabling the user to explore the data interactively and make changes to most of the parameters without long delays. This threshold is aimed at keeping the stress for the user at a minimum. An even higher frequency is needed to ensure that the user can track objects in a visualization that is animated either by progressively changing the viewpoint or because the data inspected is time-dependent itself and thusly rendered ('fusion frequency', 20Hz).

2.2. Data Types

Depending on the simulation or the measurement the data originates from, its structure and shape can vary significantly. It will usually be classified using the attributes dimensionality, type and structure. The data type can be a scalar, a vector or a tensor depending on whether a single attribute, n attributes or $n \times m$ attributes are present. The dimensionality indicates the number of different attributes, regardless of their type. *Multivariate* (instead of just *multidimensional*) data is meant to signify attributes that need not necessarily be correlated. Examples for attributes from simulation as well as measurement can be position, temperature, pressure, velocity (vector), flow direction (tensor) etc. The structure of the data can either be grid-based or gridless. Grid-based data implies the availability of connectivity information between points, and there is a corresponding taxonomy further classifying them into structured and unstructured variants (see figure 2.2). Structured grids contain the connectivity implicitly, while unstructured grids require it to be defined explicitly, thus needing additional storage space.

Structured grids can be further differentiated into *uniform*, *rectilinear*, and *curvilinear grids*. In a uniform grid all cells are of the same size and shape, while a rectilinear grid allows for variable sizes in all dimensions, so the resulting coordinates can be calculated using on grid function per dimension. In curvilinear grids all vertices need to be stored individually. The only relevant variant for this work is the basic uniform grid holding tensor data, as it is obtained from MRI scans on live subjects. Its structure can be defined by the following for-

mula for the grid vertices:

$$v_{i,j,k} = \begin{pmatrix} i\Delta x \\ j\Delta y \\ k\Delta z \end{pmatrix} \tag{2.1}$$

where $\Delta x \times \Delta y \times \Delta z$ is the size of a single cell.

Unstructured grids are not in the scope of any of the proposed algorithms and thus will not be described in further detail, besides stating that *simplex grids* basically consist of triangles or tetrahedra while *zoo grids* contain a number of different primitives, thus requiring algorithms that adapt to the geometry of the current cell.

Gridless data is also referred to as *scattered data* and will be mainly used in the context of this dissertation. Here all the attributes are tied to a vertex with explicit position in $\mathbb{R}^n$ and lacking any connectivity information. Examples for this kind of data range from simple particle systems, or direct point cloud visualization [HE03], over different information visualization approaches of high-dimensional data [And72, vWvL93, STDS95] to *radial-basis-function*(RBF)-based visualization [JWH+04], where the vertices are used as RBF centers.

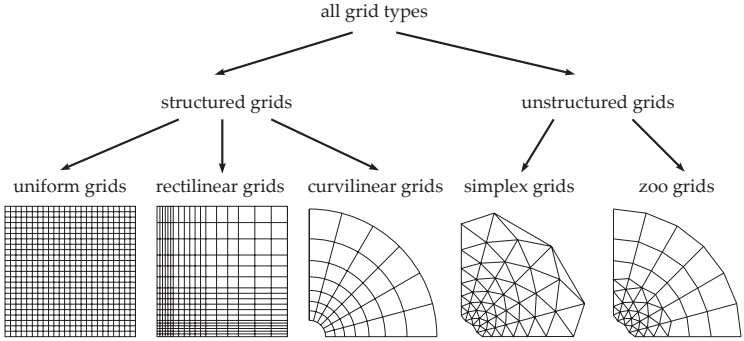


Figure 2.2.: Classification of grid types.

2.3. Graphics Hardware

One major focus in this work is the utilization of graphics hardware for the optimization of parts of the visualization pipeline. Programmable GPUs are nowadays widely available and do not anymore bear the price tag of the special rendering hardware as manufactured in the 1990s by SGI [MBDM97] or Evans

& Sutherland but still offer many times the performance of these solutions. Consumer GPUs have even evolved so much as to be nearly as flexibly programmable as the CPU and employ a fully floating-point-precision pipeline, which caused a paradigm shift away from pure graphics co-processors to general computation aids. Additionally, the low-level assembler programming interface has been superseded over the years by high-level languages like CG [MGA03] or GLSL [KBR06].

2.3.1. The OpenGL Pipeline

Similarly to visualization itself, the generation of images from data also follows a pipeline concept. A 3D scene usually consists of a collection of primitives, like points, triangles, and quads and composite geometry generated through concatenation of the basic primitives. These primitives are passed through several stages in a pipeline [FvDFH90] and result in a rasterized pixel image. The input of the pipeline are always vertices with an associated color, normal, texture coordinates, etc. that are later assembled into the required primitives, which are in turn rasterized into a pixel image that will then be output on screen. In the following the pipeline of the OpenGL API [Opeb] will be described, but all concepts work analogously in DirectX [Bly06].

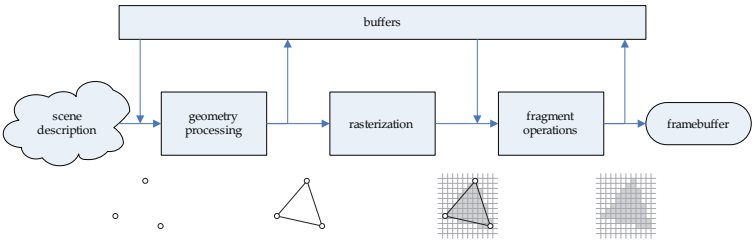


Figure 2.3.: The OpenGL pipeline and its main stages.

The first major stage in the pipeline is the geometry processing. Formerly this was also known as the transform & lighting (T&L) stage according to its functionality in the fixed-function pipeline and was first hardware-accelerated in a consumer product in the original Nvidia GeForce chip. Since the advent of Direct3D10-generation (D3D 10) hardware, two flexibly programmable substages have taken its place: a vertex shader and an (optional) geometry shader. The first receives a stream of attributed vertices in their local object space. Then, according to the fixed-function functionality, the following sequence of computations takes place:

The model transform, which specifies the location and orientation of an object with respect to the world space. In OpenGL this is always combined with the view transform, which encodes the observer position and orientation, resulting in coordinates in eye/camera space. DirectX, on the other hand, allows to separate these two transformations. Usually the transformation is formulated as a 4×4 matrix in homogenous coordinates. As after this calculation the viewing direction as well as the normals are known in the same coordinate space, per-vertex lighting can be performed. With programmable hardware, this legacy functionality can be replaced or also extended by other computations that are valid per vertex and can be linearly interpolated between them to take some load off the later stages. The projection transform maps the contents of the so-called *view frustum* into the *canonical view volume*, to be then divided by the homogenous coordinate (*perspective division*) and resulting in coordinates between -1 and 1 in all three dimensions. Finally the *viewport transform* provides the screen-space pixel coordinates for addressing the frame buffer.

The so-called *primitive assembly* is also part of the geometry stage. The primitive type that has been selected when passing the vertices to the graphics API will be created at this point. The choice can be used to reduce the number of uploaded vertices, since for example every primitive has a corresponding *strip* variant, which in case of a triangle strip only requires one additional vertex after the first complete triangle to complete another one.

The newly introduced second programmable stage of the geometry processing is the geometry shader. It receives the assembled primitives as input (optionally with neighboring vertices in the case of strips or fans) and is executed once per primitive as defined by the user. This shader is intended for generating further geometry from that input, for example by smoothly subdividing an input geometry depending on its distance to the viewer, or generating parameterized geometry from a concise representation to conserve bandwidth between CPU and GPU. A simple example would be a tesselated tube computed from a line strip (its 'backbone') directly on the GPU. It also allows for the destruction of vertices by just omitting them from the output. In contrast to fragments, before DirectX 10, vertices could never be explicitly removed from the graphics pipeline once inserted.

Rasterization converts the incoming primitives into fragments, which can be seen as proto-pixels with a viewport position, a depth, color, and additional attributes like texture coordinates etc. In this stage the per-vertex attributes are also interpolated. In the fixed function pipeline, the textures were accessed and applied, however this functionality is also available in the fragment shader. Such a shader is executed once per produced fragment and allows for complex surface shading and effects and will often have the highest computational requirements of all stages, if only for the fact that a single triangle is likely to produce more fragments than the three vertices it consists of (which is why the latest GPU architectures with the according structure had a relation of 1:3

between vertex shader and fragment shader processing units). Before committing the fragments to the framebuffer, several tests are executed: alpha, depth and stencil test. The first discards fragments that do not meet a user-definable minimum requirement for opacity. The second discards fragments that have a higher depth than the fragment already present in the framebuffer and would thus not be visible anyway, while the last checks fragment positions against a user-definable mask in screen space.

Fully processed fragments pass through the last stage, *alpha blending*. Here different operations are available depending on the alpha value of the written fragment and the fragment already in the framebuffer. The most common blending mode is realized using the so-called *over operator* [PD84], which is used for rendering semi-transparent objects (A and B) and works as follows:

$$C_{out} = \alpha_A C_A + \alpha_B C_B (1 - \alpha_A) \quad (2.2)$$

$$\alpha_{out} = \alpha_A + \alpha_B (1 - \alpha_A) \quad (2.3)$$

where C is the color and α corresponds to $1 -$ [transparency fraction].

It is important to note that as of D3D 10, all stages can render into a buffer and fetch their input from a buffer as well. This allows for multiple, complex feedback loops on the GPU itself to preprocess data that has to be rendered and postprocess images before writing them to the frame buffer. The main advantage besides the algorithmic flexibility is the avoidance of unneeded traffic between CPU and GPU. Also by now all of the shader stages have the same features and instruction sets – except the special instructions used to generate vertex attributes and emit a completed primitive in the geometry shader. This in turn has led to unified shader architectures, where the processing units are not statically assigned to any particular stage – as was common still with DirectX9-level hardware – but can in theory be dynamically distributed to allow for load balancing depending on the currently running algorithms.

In addition to the described features OpenGL also offers an imaging pipeline and evaluators for the generation of parametric surfaces. Since GPUs and drivers have abandoned implementing any subset of this functionality in hardware basically ever since SGI has no influence on graphics hardware technology anymore, these features are not fit for accelerating or otherwise significantly improving visualization algorithms and are therefore not covered.

2.3.2. Programming Paradigm and Shaders

What mainly separates programmable GPUs from general-purpose CPUs nowadays is the restricted data flow and data organization. The vertex shader always maps input vertices to output vertices with a 1:1 ratio. A vertex can have several attributes, whose number can also arbitrarily change between input and output, but it is not possible to remove a vertex from the pipeline at this

stage. The only work-around to achieve a similar effect is to position it at the homogenous coordinate 0, that is, at infinity, which will cause it to be clipped later on. Additional data can be accessed in the form of textures. To implement data structures that are more complex than a simple nD array with $1 \leq n \leq 3$ cross-references are needed. In practice that means to employ one or more index textures which in turn yield coordinates that are used for the so-called *dependent lookups* into other textures and so on. A generic approach is shown in [LKS+06], which also summarizes a number of GPU-adapted data structures published previously. The geometry shader only has the additional capability of emitting a limited number of attributes over multiple vertices of one or more primitives chosen from among those available in OpenGL. Thus, in this stage an *n:m* mapping is obtained which can optionally extend as well as reduce data. In the last programmable stage, the fragment shader, only single fragments are processed, which can also be removed from the pipeline (*killed*), but not relocated (in contrast to the other two shaders, where the focus is on coordinate transformations), or added. Note that the *input:output* ratio is always a function of the destination resolution, as the rasterization depends on that resolution. This allows for reduction/expansion algorithms of any kind, for example up-/downsampling of textures, or pyramid processing of large input over several iterations etc, but not for the dynamic insertion of single fragments.

Originally shaders had to be programmed in assembler. Data could be read from pre-defined attributes (of vertices/fragments), state variables of the GL, and uniforms (per-pass constants that could be accessed directly from the CPU code). Output was only possible to pre-defined attributes as well. Temporary data was stored in numbered registers, which made the reading and understanding of such code very tedious. Over time, the maximum length as well as the capabilities of such shaders were increased. Program flow control, for example, was not available in fragment shaders from the start because it was too costly to implement the parallelization of the processing of neighboring pixels in case of differing program paths. Then high-level languages became available: Cg ("C for graphics") [MGA03], the high-level shading language (HLSL) of DirectX [Bly06], and the GL shading language (GLSL) [KBR06]. These allowed mainly for much cleaner code by automatically mapping named variables to registers, binding attributes and uniforms by name, and allowing the use of subroutines and composite data structures (locally). In addition to these computer-graphics driven developments, APIs for the usage of GPUs as numeric coprocessors, or stream processors, have also been developed to abstract the GPU interface even further. One example is a linear algebra framework for GPUs that can be used for simulation [KW03b]. Stream processing generally has the scope of SIMD (single instruction, multiple data) operations on a long list (stream) of uniform data and can be used for image processing, for example. Two approaches exist that make GPU-accelerated parallel math

available directly in standard C/C++ programs and thus more easy to use: Brook[BFH+04] and Sh [MQP02]. Recently, Nvidia has brought forth an 'official' solution called *Compute Unified Device Architecture*(CUDA) [CUD], which also uses an additional compiler pass to separate GPU and CPU code. Nvidia supports its own approach by specifically offering Quadro-like CUDA-enabled hardware without monitor connectors at a lower price (named Tesla).

2.3.3. Volume Visualization

Volume rendering comprises a multitude of algorithms that render grid-based scalar data in a 3D domain. The scalar values usually represent the density at the corresponding point in space. Typical examples include sensor data as generated in the medical context, for example by CT (*computer tomography*) scanners or MRI (*magnetic resonance imaging*) scanners. Other sources are simulations with discretized domains, like fluid simulations, or point clouds that are quantized and binned for performance reasons, thus yielding a density per sample point. The resulting grid of scalar values is figuratively subdivided halfway between the sample points, resulting in rectangular cells, which are also called *voxels* (a portmanteau of volume and pixel) and have one sample located at their center. Usually the rendering does not represent the scalars directly, but transforms it first via a *transfer function*, which maps the density to color and opacity $t : \rho \rightarrow (r, g, b, \alpha)$.

Volume visualization can be categorized into indirect and direct methods. The former first extract some kind of intermediate structure from the data which can be rendered faster than the whole volume. The most prominent example is isosurface rendering, which extracts all points whose scalar value equals a user-defined value. The rendering is then accomplished based on polygons resulting from the tessellation of the resulting surface via an algorithm like the classical *marching cubes* [LC87]. Alternatively, the surface is extracted on-the-fly while rendering with a direct method [Lev88]. This approach obviously incurs in a loss of information (the remaining iso-values), and thus of the context of the selected surface, which for example in medicine is a significant problem (what value does the display of a tumor have if the doctor cannot see how it affects surrounding tissue or bones).

Direct methods try to circumvent this by rendering the data as a whole by default. They treat each sample point as self-illuminated and opaque, according to the density found in the data. The so-called *volume rendering integral* has to be solved to accumulate the emission/absorption of each point along viewing rays emitted from the view point. In the following only a short introduction on direct volume rendering methods will be given as a foundation for the visualization in chapter 4.

The foundation for volume rendering is the light transport theory neglecting any scattering effects [HHS93]. The integral that has to be solved for every

pixel of the frame buffer describes the intensity $I(x)$ at a position x on a viewing ray through the respective pixel:

$$I(x) = I_f e^{-\tau(x_f,x)} + \int_{x_f}^{x} \eta(x') e^{-\tau(x',x)} dx', \quad \tau(x_1,x_2) = \int_{x_1}^{x_2} \kappa(x) dx \tag{2.4}$$

where I_f is the (scene) intensity at the far end of the volume x_f, τ the optical depth (the quantity of light removed from a beam depending on the length of its path by absorption), κ the absorption coefficient and η the emission.

If the emission of each voxel C_k and the respective opacity α_k, as derived from the transfer function t, are used to model η and τ, the integral can be simplified and discretized to the following compositing formula

$$I = \sum_{k=1}^{n} C_k \prod_{i=0}^{k-1} (1 - \alpha_i). \tag{2.5}$$

The approximation of τ is just the product of the opacities of the voxels that the integration has already passed, the actual values of C_k, α_k are interpolated from the surrounding samples (as it is unlikely that the cast ray will actually intersect the voxel centers).

Since volume rendering is employed as a means to achieve frame rates that are independent of potentially too large data in the context of this dissertation, image-space methods, whose cost is thus proportional to the output resolution, were the first choice. Object-space methods, such as splatting [Wes90] and cell projection [ST90] will not be described in detail. The classic image-space approach is raycasting, i.e. an implementation very close to what has been established above [Lev88]. With programmable GPUs it has also been implemented in hardware recently [SSKE05]. Since the availability of graphics cards capable of multi-texturing (reading from more than one texture per fragment) and dependent lookups (for the transfer function), texture-based volume rendering at interactive frame rates has become feasible. It was first presented for graphics workstations by SGI [CCF94], but has been available on consumer graphics cards for several years now [RSEB+00]. It renders the volume making use of a textured proxy geometry rendered back-to-front[1] and exploits blending for the accumulation of the volume rendering integral. Using 2D textures, three axis-aligned slice stacks are employed, which are more prone to artifacts at the edges where one would look 'between' the slices if spaced too far apart. With 3D textures, the slices can be more conveniently placed parallel to the viewport. Many improvements of the basic algorithm exist, among those pre-integration, which adds inter-slice pre-computed integration [EKE01].

[1] and thus should be considered a hybrid image/object-space approach although the main load is on the texture units and fragment processing analogously to pure image-space algorithms

2.4. Human-Computer Interaction

An all-too-often disregarded area of computer science is Human-Computer Interaction (HCI), which is essential for enabling the user to influence the visualization pipeline – if there are parameters, the user must be able to inspect and influence them efficiently. HCI studies the effects of user interfaces on human psychology and behavior and vice versa. The ultimate goal is the derivation of rules and development of methods that ensure that software and hardware are designed with the user in mind such that computer usage becomes as efficient and comfortable as possible. Evaluation by user studies also constitutes a central component of this area [HBC+92]. Technical constraints for these studies are set by the available hardware (display devices, input devices) and are periodically tried to be overcome by proposing new devices that improve on existing ones or deal with very specific problems, like six-degrees-of-freedom devices (for example the PHANTOM device [Pha]) or portable mouse replacements for 2D interaction [BSW07] or 3D interaction (the 'Space Mouse'). Human factors are being researched for quite a while and deal with the limitations of human perception (physiological constraints) [WS82] or the human brain [Mil56]. Several authors have thus come up with recommendations and rules that should be observed when designing user interfaces. One Example is the concept exposed by Norman [Nor88], who describes interaction problems making use of the human action cycle with its three main stages, the goal formation (What do I want to do?), execution (What tasks are there? In which sequence do actions need to be performed?), and evaluation (What happened? Does this comply with my intentions?). Another take on the problem are the eight 'golden' rules by Shneiderman [SP92]. Part of these concepts have also been passed as a standard for dialog design, the DIN ISO 9241. The standard requires the following principles to be observed:

1. suitability for the task: the user should be able to realize his intentions without being impeded (*goal formation*)
2. self-descriptiveness: it must be obvious which interactions are possible and required to achieve a goal (*gulf of execution*, i.e. the discrepancy between the actions the user supposes he has to take and those that really need to be taken)
3. controllability: the user controls when and with which speed partial operations are to be executed (*gulf of execution*)
4. conformity with user expectations: effects are consistent with previous experiences (*goal formation* and *gulf of execution*)
5. error tolerance: wrong input is admonished and correctable without losing work (*gulf of execution* and *gulf of evaluation*, i.e. the user's difficulty of determining the state of a system)
6. suitability for individualization: the user can adjust the interaction pos-

sibilities according to his experience and preference, i. e. trading speed for self-descriptiveness etc. (*gulf of execution*)

7. suitability for learning: the system allows the user to memorize the interactions necessary by repetition since the affordances are placed consistently and perform consistently (*gulf of execution*).

The standard however does not seem to stress feedback as much as both Norman and Shneiderman do, that is, that any software/hardware must always notify the user of its current state, whether action is required or even allowed and what the outcome of a pending request is. This is only partly covered by point two in my opinion.

3 Visualization of Point Clouds

Point clouds, i.e. datasets that exhibit an explicit position in $\mathbb{R}^n$, usually with $1 \leq n \leq 3$ (and also $t \in T$ in the case of time-dependent datasets) in addition to a limited number of other attributes, can be intuitively placed in 3D space and visualized. A point can be considered infinitely small and rendered in a way as to occupy only one pixel on the screen, thus effectively visualizing only its position and a very limited number of distinguishable colors [Hea96]. An alternative is to place some kind of geometry at its location, in the simplest case resulting in a screen-space splat [RL00] with the shape of a circle (as happens with smoothed points in OpenGL, for example). Such circles, and also real spheres, can already intuitively display two additional dimensions by mapping them to the radius and surface color. Such symbolic parametric objects that visually represent the attributes of a data point are referred to as *icons* or *glyphs*. The goal of rendering a dataset with icons is to have a clearer, compact representation of large datasets which often effectively are tables containing numerical values only. The relations between attribute values should be easier to comprehend, thus generating a more meaningful output for the user [PPWS95]. Attributes can be canonically mapped to visualization parameters like color, size, or shape with varying implications for the sensible range and discernibility. However, the more complex a parametric geometry becomes, the more screen space or the higher a resolution is required to enable the user to distinguish all the different features of the geometry. Consequently, the upper bound for the number of features a certain glyph can represent in the chosen context is reached when the visualization irrevocably forfeits clearness, compactness or meaningfulness. Another limit for this technique is the number of simultaneously renderable glyphs in terms of screen space as well as performance, since complex glyphs can increase the geometry load of a scene enough to make interactive visualization challenging [Rag02] when not taking advantage of *level-of-detail* (LOD) techniques or the like.

This chapter will elaborate on the efficient GPU-based rendering of basic primitives and introduce different new application-specific glyphs that have been

The work in this chapter is based on [RE05a], [GRVE07], [GRE09], and [RBE+06]. Some paragraphs and figures are excerpted from the original publications.

tailored to compactly represent the most important attributes of data from particular applications in thermodynamics and medicine. While forgoing at least part of the generality of the proposed solution for the last two glyphs, the specialization makes it easier for experts from the field to comprehend the datasets [PvW94].

3.1. Related Work

The visualization of points is an area of research that by now is getting a lot of attention, which can be understood when looking more closely at the widening scope of the published work. Besides the canonical use for basic scatterplots, this topic should actually be differentiated by the semantic density of the points themselves. One branch of research considers the points as individual entities which do not necessarily have any kind of relation to each other, and have all attributes on their own, which results in a high semantic density, often accompanied by rendering primitives far more complex than a single pixel. The other branch employs the points only as a part of an object, usually its surface (therefore also referred to as *point set surfaces*). This means that for the latter points are only a means for representing a surface and will thus rarely exhibit semantics that go beyond the spatial parameters (position, orientation, extents). Connectivity information is not necessarily stored or even available.

Point-sampled geometry can be acquired using laser scanners [LPC+00] or digital cameras, for example in conjunction with "structured light" (projected patterns) [SWPG05] or by using a controlled environment with adjustable matte and configurable light positions [MPZ+02]. Rendering is usually performed using an elliptical footprint (also called *surfel* [PZvBG00]), which can be considered a simple glyph as well. The rendering of the processed and reconstructed surfaces has evolved quite a lot in the past years. The basics were laid with CPU-based algorithms for anti-aliased splatting [ZPvBG01] and hierarchical splatting with a very efficient data structure [BWK02]. GPU-accelerated algorithms range from very fast projected splats without shading [BK03] (also including clipping [GP03]) over perspectively correct Phong-shaded splats with clipping [BSK04] to well-performing per-pixel shaded splats exploiting deferred shading [BHZK05].

All the publications which elaborate on points as distinct entities with a variable number of attributes tend to employ 3D primitives rendered as single points or billboards, and compute the final glyph shape on the graphics card with algorithms of varying precision and complexity. The basic approach results in depth-correct (in case of an orthographic projection) texture-based spheres [BDST04]. More computationally intensive approaches include different implementations for ellipsoids, either using quad geometry [Gum03] or point geometry [KE04]; one article also proposes other types of quadrics (see

also [Zwi95]) as primitives [TL04].

Point-based rendering is also exploited for subsampling scenes too large to render in a conventional way. One approach generates samples randomly for fine (subpixel) details and renders large polygons directly [WFP+01]. This approach has also been extended for animated scenes [WS02].

An introduction to the application domains as well as the characteristics of the typical datasets will be given in the next section; the rendering algorithms are illustrated by examples from these fields.

3.2. Molecular Dynamics Data

Nowadays, simulations on a molecular level are gaining more and more importance in research areas like chemistry and thermodynamics, materials and several other areas where nanoscale particles are of importance. Such simulations are very expensive in terms of computational power, but with commodity-off-the-shelf clusters becoming more common and filling this gap at relatively low cost, the simulations are becoming larger in terms of particle numbers as well as simulated time. The data generated by such simulations is only limited by the available processing power, currently including hundreds of thousands of molecules and thus posing a challenge for the storage as well as for those striving to interactively visualize the results.

Simulations bridge the gap between theory and experimental practice, making it possible to verify the theoretical models on the one hand and on the other hand replacing experiments under difficult conditions, like extremely small scales [AT87] or metastable state. They also open up the possibility of probing values like energy/velocity or distance of all the atoms and thus give researchers access to many more parameters to help them understand chemical or physical effects. Simulations on a molecular level have been successfully employed since the 1950s, however, especially on the nanoscale level, many effects are not yet completely understood.

In the context of the project 688[1] as well as during our collaboration with the Institute of Thermodynamics and Thermal Process Engineering (ITT) and the Institute of Technical Biochemistry (ITB) suitable visualization methods for a large number of atoms have been developed. Researchers from both fields rely heavily on the use of molecular dynamics simulations to investigate the behavior of organic and anorganic matter at the nanometer scale. Biochemists research the behavior of proteins in solvent, for example, while thermodynamicists are interested, among other questions, in the condensation process, that is, the nucleation of vapor into liquid phase.

[1] Landesstiftungsprojekt "Visualisierung der Keimbildung in Mischungen für skalenübergreifende Modelle"

The key properties of nucleation processes are the nucleation rate and the critical cluster size. The critical cluster size designates clusters with a certain number of monomers 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. While the classical nucleation theory can be used to predict nucleation rates, it is not correct for many practical cases and fails completely when mixtures of several substances are concerned. This is why molecular dynamics simulations are used instead to predict the nucleation rates more accurately. This method has the drawback, however, that a large number of molecules has to be used to get meaningful results. In practice, the number is limited by the processing power available to the researchers, but systems with more than 10^5 molecules are not uncommon, for distributed simulation runs 10^7 molecules and more are feasible. At the ITT the Lennard-Jones potential model [LJ31] is employed for computational efficiency and also visualization, simplifying molecules down to few positions and their respective potential, usually displayed as spheres. The datasets are very complex all the same because of the sheer number of molecules involved (see figure 3.1).

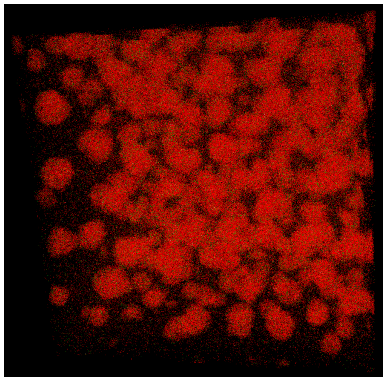


Figure 3.1.: Nucleation simulation consisting of 1.3 million CO_2 molecules.

Biochemists, on the other hand, do abstract from the single atoms. They investigate the behavior of proteins in solvent, which means the motion of the protein itself. It is now evident that protein flexibility and correlated motions [DV06] are essential for protein function. Of all possible shapes of a molecule the most interesting ones are those which are local minima in terms of potential energy. These local minima are called conformations. Hence, conformational behavior of a molecule is the process of transitions between individual conformations of the molecule around the energy minimum. As in thermo-

dynamics, molecular dynamics simulation is one of the principal tools in the theoretical study. Datasets from this research area have a much lower number of molecules, in the range of 10^4 to 10^5. Commonly these datasets are visualized per atom, for example with the so-called spacefill style, which employs one sphere per atom using the Van-der-Waals radius as size, or using the ball-and-stick style, which draws much smaller atom spheres and adds cylinders between bonded atoms (see figure 3.2).

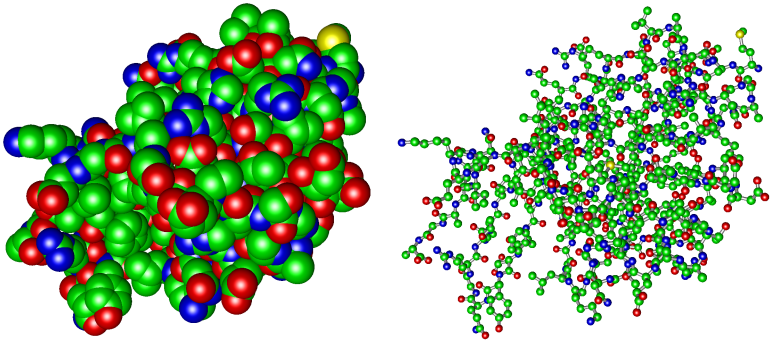


Figure 3.2.: Spacefill and ball-and-stick rendering modes for the 1OGZ protein

.

3.2.1. Visualization of Molecular Dynamics Data

The visualization of molecular dynamics data has been tackled with many a solution over the years, all of them radically different in approach, features, and performance. The most widely spread tools are probably Chimera [Chi], PyMOL [Pym] and VMD [VMD]. Some packages have been used for such a long time that they have changed names several times or are integrated into different systems, like RasMol [SMW95], which has evolved into Protein Explorer [Mar02] in Chime and FirstGlance in Jmol [Jmo], or WebLab-Viewer, which is superseded by Discovery Viewer/Studio [acc]. Generic visualization packages, like AVS [AVS] or amira [ami], also come with special modules for molecular visualization nowadays. All of these programs offer a functional rendering output and a varying range of analysis options mostly for biochemists, and some even have an integrated simulation core, like BallView [MHLK05]. The rendering performance of these tools, however, does not allow for interactivity when rendering very large molecules (>50.000 atoms) since most of them use polygon-based representations, but no LOD techniques. Another salient weakness of these tools comes into play when dealing with time-based data, which is usually loaded into memory as a whole

– the user also has to wait a considerable time until the dataset is fully loaded and can be navigated from start to end. The dataset size is thus also limited to the available memory and prevents large simulations to be loaded at all. For the specific rendering of protein molecules there is also the very popular solution of using Java applets, of which there are a multitude: KiNG [KiN], Jmol [Jmo], WebMol [Web], Protein Workshop [MGB+05], to mention just a few. The performance (and memory requirements) of these however cannot compare to native-code applications.

Recent research, however, has also shown approaches that allow to overcome the performance problems. Max [Max04] has proposed a splat-based approach for large molecules, however the rendering style has more in common with volume splatting than what researchers in the fields are used to. A solution based on depth-correct textured billboards for sphere, cylinder, and helix primitives has been presented in [BDST04]. It supports several rendering styles and offers better performance for large molecules than most of the tools referenced above, but the visual quality of the rendering can still be improved, since the limited resolution of the normal/depth textures will cause artifacts (see figure 3.3). Another solution [HOF05] supports texture-based as well as polygon-based rendering including LOD techniques for whole-molecule surfaces. The suitability of these tools for time-based datasets, though, is unclear at best.

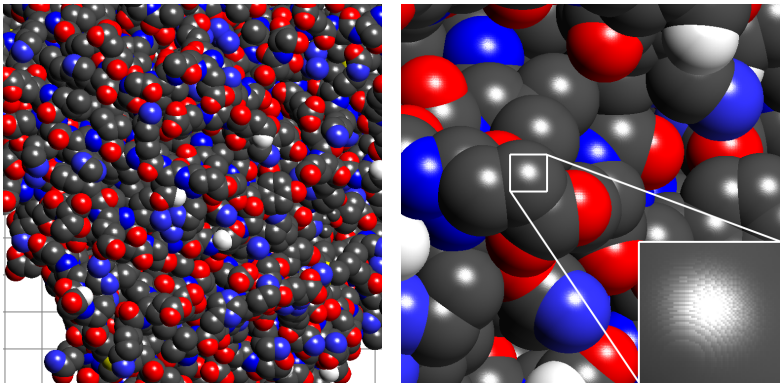


Figure 3.3.: Perturbed normals caused by precision issues when using normal textures for sphere billboards. Screenshot taken of the freely available TexMol [BDST04].

To improve the depth perception for the visualized data, stereo output and VR hardware like powerwalls or CAVEs have been employed. There have been several approaches which visualize simulation results in a VR environment and optionally allow simulation steering if the number of simulated par-

ticles is not very high [AF98, SGS01], but without employing particular techniques to allow for the rendering of large numbers of molecules. However, there is some previous work on visualizing massive data sets of more than a million molecules by using multi-resolution rendering to ensure interactivity [NKV99].

3.2.2. Point-based Glyphs

Visualization of large molecular datasets with polygon-based glyphs is not the most convenient solution for the current implementation in hardware as well as in software, as can be deduced from the related work: if a polygon-based algorithm is to be used, the tessellation must be adapted depending on the rendered size to obtain smooth surfaces and not degrade performance below 5-10 frames per second ("interactivity"). This can be achieved taking advantage of level-of-detail methods [FS93, Lak04] to adapt the number of polygons as to guarantee smooth surfaces for near objects and coarsely render distant ones to save performance. However a large number of polygons has to be sent to the graphics card in any case. Taking a look at the related work from point-based visualization and molecular visualization, it becomes evident that the results from one field can be successfully transferred to the other to overcome the deficits of the approaches conventionally employed. GPU-based algorithms for the generation of glyphs are needed, which will be described in the following sections. The implications of time-based data must be considered regarding the technical level as well as the subsequent usage of the visualization to gain additional insight must be kept in mind from the human-computer interaction point of view. This will be elaborated upon in Section 3.2.4.

Generating parametric surfaces using fragment shaders on the GPU has been investigated in several publications. Two different approaches implemented Lindenmayer Systems [Lin68] to explicitly generate geometry in vertex buffers, in one case for subdivision curves [MP03], in the other using the example of procedurally generated trees [LH04]. The complementary approach of raycasting implicit representations of surfaces directly onto a proxy geometry has been implemented for ellipsoids with optimally sized quads [Gum03] and with single points to save on bandwidth at the expense of fragment shader load [KE04]. The raycasting method has also been chosen in the context of this thesis, on the one hand because of the high rendering quality, also compared to texture-based approaches, where applicable ([BDST04], see section 3.2.1), and on the other because of the high performance which can be obtained when every stage of the OpenGL pipeline is carefully optimized.

To render any glyph surface on the GPU using a billboard geometry, be it a quad, a triangle or a single point, it must first be adequately parameterized to be as bandwidth-efficient as possible, otherwise the whole point of cutting down on polygon transfer is missed. The attribute with the the biggest poten-

tial for space saving usually is the position of a glyph, note however that the higher the number of additional attributes per point becomes, the more the impact of the position diminishes. The approach proposed by Hopf [HE03], for example, quantizes positions to three bytes and uses a hierarchy to reconstruct most of the original positional precision. Previous work even goes as far as using only 13 bits for the position [RL00]. This special case, however, has the advantage of working on contiguous points representing a surface, resulting in dense groups of relatively many points.

To put the issue into perspective for the MD datasets used in this thesis, the resulting point clouds have a less advantageous spatial distribution compared to the surfaces because the points can usually not be relied upon to be as tightly packed (many of the data sets contain gases). Quantization, though, is sensitive to large empty spaces since the positional precision decreases proportionally to the maximum extent of the spanned space. Considering a one-dimensional example with a bounding box of size s_{BB} and $b = 2n + 1$ bins (to ensure that we have a zero bin and equal resolution on both sizes of that center), the maximum positional error is half of the bin size $\Delta_x = s_{bin} = s_{BB}/2b$. The maximum relative error for a unit length and 8 bits for positioning consequently is $\delta_x = 1/510 \approx 2‰$, which looks more than acceptable (in 3D, this error would increase to $\sqrt{3}\delta_x \approx 3.5‰$). In a molecular simulation, however, one has to consider that the error has to be sufficiently small as not to generate positions which make two Van-der-Waals radii visibly overlap, because that would reflect a state which should not be obtainable if the simulation is implemented correctly. Actually, current implementations (in thermodynamics, for example) go as far as allowing a minimal overlap, however the goal of the visualization should still be to match the data as precisely as possible, and since the most influential situation that can easily lead to false conclusions is such an overlap, it is to be minimized. In an example scenario of supersaturated gas with one million atoms of radius 1 and about seven times as much empty space, the simulation domain would be 200^3, with $s_{BB} = 200$. Allowing at most 5% overlap, the most misleading case would be two touching atoms, both mis-positioned by the maximum error, resulting in an overlap of $2\Delta_x \triangleq 0.05$. These prerequisites require 2000 buckets per axis. It is not possible to address them using merely 8 bits, however a second hierarchy level will already result in sufficiently precise positions.

Data Transfer Optimization

Quantized data helps keeping an application's memory footprint small and is as such a viable option, however uploading quantized data to the GPU does not always benefit the performance of applications to an equal degree. To more precisely investigate the effects of quantization and different uploading and parametrization strategies, several benchmarks have been conducted,

the parameters of which can be seen in table 3.1. The results are discussed in the following. As in the remainder of this dissertation, the performance is measured using an OpenGL-based point rendering implementation under WindowsXP. A random subset of tests has been implemented using DirectX 9 only to find a nearly equal performance (within a margin of 3%), the same is valid for the original implementation when compiled and used under Linux, so no further details will be given and the overall result can be expected to be valid more generally. Since PCs implementing the x86 architecture still can differ in performance because of differences in memory support and system bus capabilities, few but reasonably different systems have been chosen for the tests, namely two 'fast' machines (an Intel Core2 Duo 6600 and an AMD Phenom 9600) and a 'slow' and a 'legacy' system (AMD Athlon64 X2 4400+ and an Intel Pentium 4 2.4GHz, which is the only machine that only supports AGP graphics and is mostly used as baseline reference). Four PCIe cards were rotated through the modern systems, an Nvidia GeForce 6800GS, a 7900GT, an 8600GT and an 8800GTX. It must be noted that the employed driver version (169.21) penalizes the Core2 Duo machine, since although all of the test PCs only had a single monitor, using the driver in 'multiple display performance mode' (which is the default) instead of 'single display performance mode' cost only the Core2 Duo machine about 10% performance, while the AMD-specific code path in the driver seems to be insensitive to this option. No AMD/ATI-based graphics cards were tested since their OpenGL support is currently insufficient for any of the proposed algorithms. Error bars in the presented diagrams are aimed at highlighting methods that suffer from high performance fluctuations and thus should only be employed when no better alternative is available.

The first question that needs to be answered is how beneficial the reduced bandwidth requirements of quantized data are when uploading vertices to a graphics card. In figure 3.4 it can be seen that there are cases where the performance of GPU-resident point clouds (unsuited for time-based data) can nearly be matched using signed-byte quantization for uploads, proving the general validity of the approach. However for currently available high-end cards this is not universally valid. In some cases the maximum performance for shorts can even be higher (see especially figure B.4), although twice as much data needs to be uploaded. This might be due to alignment problems – assuming the hardware is optimized for handling dwords. The more important information in these diagrams is that using shorts at least 90% of the signed-byte performance can be achieved, which is a small trade-off considering the benefit of twice the precision. This holds true for all tested combinations of CPU, chipset and GPU. Along the same lines, when using floats, even though the data quadruples, the performance only halves, which makes full-precision uploads not as taxing as one would expect. One possible explanation for this fact could be alignment issues, which need not be taken care of with wider data,

benchmark	parameters	figures
common	512^2 viewport, release build 32 bit, Nvidia driver version 169.21 with default settings and Vsync off	-
quantization	dequantization and ModelViewProjection transform vertex program, constant color fragment program, one pixel per vertex, positions randomized	3.4-3.6, B.1ff
upload points	ModelViewProjection transform vertex program, constant color fragment program generating one pixel per vertex, vertex positions on a regular grid	3.7
upload spheres	optimal bounding box calculation in vertex program, sphere glyph fragment program, vertex positions on a regular grid, color set once	3.8
parameter texture	bounding box in vertex program, dipole glyph fragment program, vertex positions on a regular grid, 500,000 vertices, color set per glyph	3.9
cylinder bounding geometry	optimal bounding box calculation in vertex program, cylinder glyph fragment program, vertex positions on a regular grid, 500,000 vertices, cylinder aspect ratio 2:1 (length:diameter), color set per glyph	3.12
overall performance	500,000 vertices with different glyph shaders and optimal bounding boxes	3.16

Table 3.1.: Benchmark parameters and modalities as used for results later in this section.

and driver optimization that might not be very thorough if byte-quantized coordinates are not used a lot by game software programmers, who should still make up the largest portion of people that work most closely together with GPU designers and the respective driver programmers. This only applies to the GeForce series, as Nvidia representatives always claim that even though the Quadro and GeForce drivers have a common core, they evolve very differently over time: driver changes for the GeForce models are only caused by game stability/performance issues, while Quadro drivers also contain bug fixes that are discovered by the developers of commercial CAD and visualization software. The latter are never integrated into the GeForce drivers if they imply performance penalties of any kind. It should finally be noted, however, that the upload modes that benefit the most from strong quantization are those with the worst overall performance, that is the different kinds of *Vertex Buffer Objects*. By design VBOs seem originally intended for multiple render passes of the same data, but the streaming variant at least should be suitable for little to no reuse. This does not seem to hold true, and this effect can be seen in all of the diagrams.

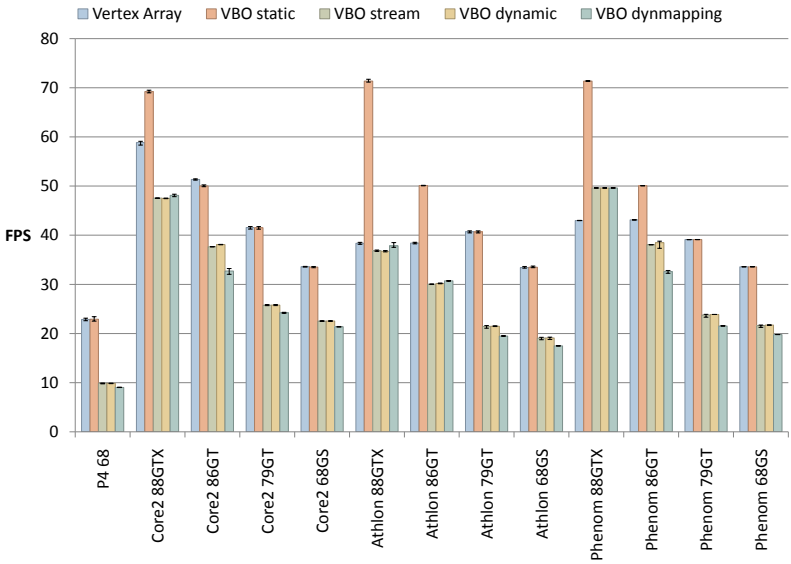


Figure 3.4.: Upload performance for 4 million byte-quantized points. In most cases vertex arrays outperform the non-static VBO variants. The Phenom shows either system performance or driver issues for vertex arrays.

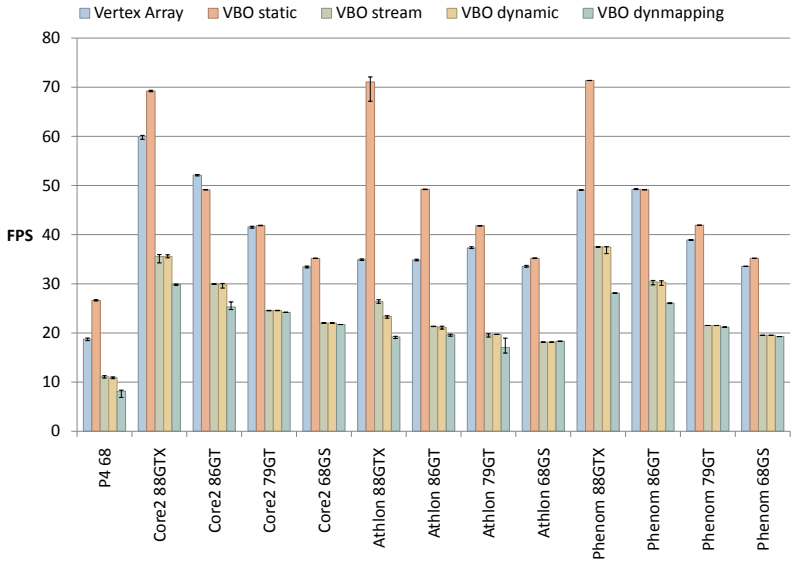


Figure 3.5.: Upload performance for 4 million short-quantized points. The performance is nearly on par with byte-quantized points using vertex arrays, while the non-static VBOs take a significant performance hit on current GPUs. In this test every setup works best with vertex arrays.

Looking at the performance figures across the different precisions, the very similar performance of static vertex buffer objects gave rise to the suspicion that such data might undergo a silent conversion on upload, however double-checking with VBOs large enough to require more than the available onboard graphics card RAM (when using floats) proved that no such conversion takes place and indeed four times the points can fit into memory when using bytes instead of floats. So the reason for this is probably a geometry processing limit, which in turn also means that the supposedly dynamically load-balanced generic computation units of the GeForce 8 can not arbitrarily be reassigned to vertex shaders, but that there must be a capped ratio, like for example 1:2 (only counting vertex shaders versus fragment shaders). Were this not the case, there should be a much higher performance gap between GeForce 8 (32 x 4 scalar computation units in total) and GeForce 7 (8 dedicated vector units for vertex processing only). On the other hand these results also indicate that the dequantization to full-precision floats when reading from byte textures effectively comes for free at least on current hardware.

The measurements also confirm some expectations since the bandwidth-invariant static VBOs gain a growing advantage the wider the uploaded data gets, however this holds also true for vertex arrays. The most conspicuous side-effect can be observed on the Athlon system: here the dynamic mapping of VBOs – that is the direct `memcpy` into what should be kernel-mapped GPU memory – is very slow even though the front side bus bandwidth of this system is second only to that of the Phenom system. The bottom line of these tests is that if the precision is at all required, using shorts offers the best performance/precision ratio in most cases.

As of OpenGL 2.0, once the precision of the uploaded data is decided, there are still several ways to upload the vertex data to the graphics card. The distinctive feature of OpenGL, compared to DirectX, is the *Immediate Mode*, which puts heavy load on the CPU, but comes with the benefit that the uploaded data need not be aligned nor linear in memory in any way and can also be computed on the fly. In contrast, the best-performing way to upload the data across all tests currently is to at least lay out every attribute linearly in memory by itself with a constant offset between values, which is enough to employ *Vertex Arrays* (see figure 3.7). This makes it very convenient to keep all data of a point locally interleaved, for example by concatenating position, color, and any remaining attributes, and then concatenating these points. The current state of the OpenGL 3.0 specification [Opec] however suggests that both of these modes will be removed, bringing it on par with OpenGL ES (*embedded systems*) and DirectX with regard to vertex upload. That leaves VBOs as the only future-proof and, at least by concept, efficient upload method. This mode forces the application to keep the data in memory uninterleaved, at least if different attribute sets are to be used (color from velocity, color from element type, etc.). Otherwise the unused attributes would have to be uploaded all the same

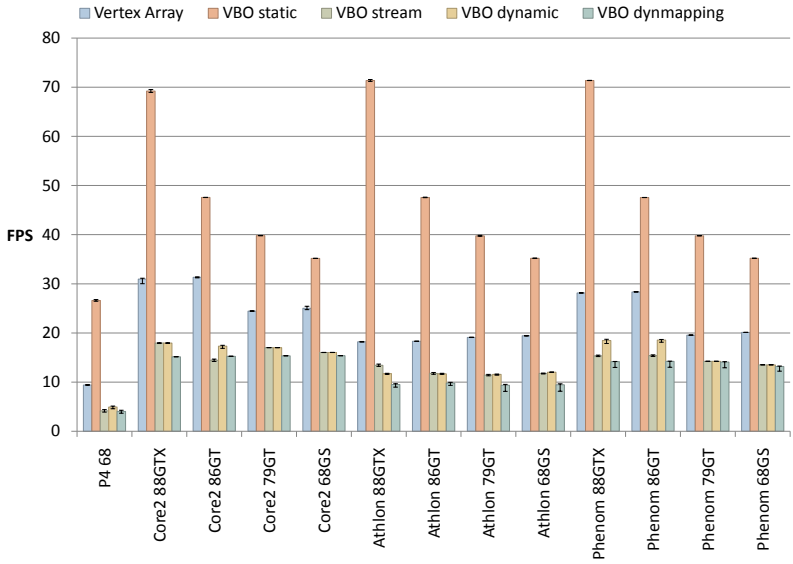


Figure 3.6.: Upload performance for 4 million unquantized points. Due to the data size, static VBOs always are much faster than the other rendering methods. In conformity with the expectations, performance is halved with respect to shorts (Figure 3.5) and vertex arrays still offer the best performance across all hardware combinations.

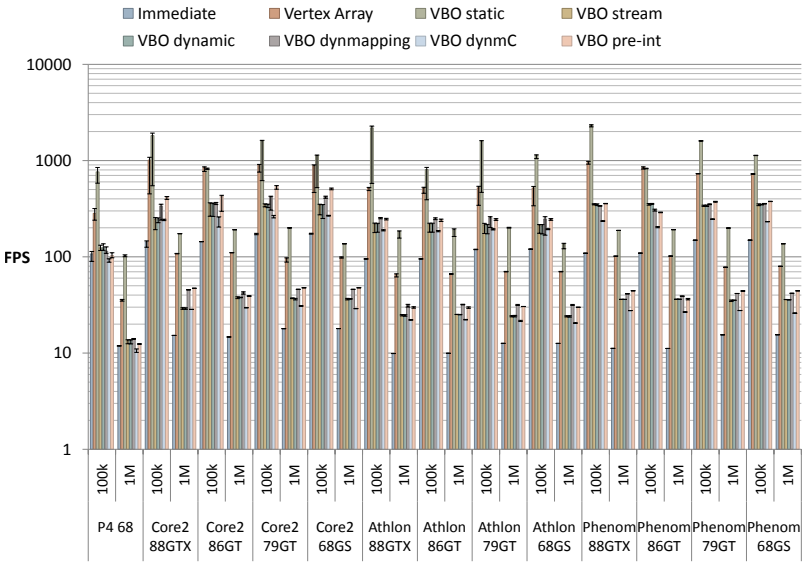


Figure 3.7.: Rendering performance of single-pixel points on different machines for all available upload modes. Vertex arrays offer the best performance for dynamic data across all machines.

or data must be uninterleaved on the fly, implying a lot of `memcpy` operations in the worst case. But even if the data is laid out to fit the rendering perfectly, that still leaves the question why VBOs are still significantly slower than vertex arrays (see figure 3.7), regardless of the specific mode utilized. The only plausible explanation for now is that VBOs might present some streaming drawback, so while vertex arrays can already be rendered while not completely uploaded to the GPU, VBOs might have some requirement before rendering can be executed for the first time. As such they might render as quickly as vertex arrays, but suffer from a delay. VBOs certainly do not look very attractive performance-wise at the moment, and if this problem is not fixed with future drivers, OpenGL 3 applications are going to suffer from a severe performance hit (at least if no multi-pass rendering is performed). However, with ray-cast GPU-based primitives, on slow GPUs the overall performance is so severely limited that the upload mode hardly makes a subjectively noticeable difference, even though the relative performance loss is significant (see figure 3.8). Surprisingly, for smaller batches of primitives the performance of the GeForce 7900 is nearly equal (for 1M spheres) or even higher (for 100K spheres) than that of the current high-end card, so there must be some overhead either in the drivers or in the architecture of the 8 series GPUs that can be detected when employing only this particular shader load, since in all other benchmarks a speedup from 7900GT to 8800GTX can be easily observed.

An effect commonly known from many benchmarks that can be found on the web (which are usually conducted using computer games, which in terms of computational costs are not much unlike GPU-based visualization applications) is that the faster a GPU is, the faster the overall system and especially the CPU has to be to significantly increase rendering performance. This is required even though much of the processing is happening on the GPU and is probably because the driver, which in the end controls the graphics card, still runs on the CPU and needs processing time. This effect can be observed when comparing the frame rates of the high-end Intel-based system to the AMD-based system. It is of course unclear whether the driver code for one of the two systems is also optimized more thoroughly than the other, however the overall performance reflects the relative CPU performance of the two systems, with the exception of the static VBO measurements, which in turn proves that this vertex data storage method really does rely mostly on the GPU and no work is offloaded silently to the CPU. This is also obvious in the quantization benchmarks: only the Core2 system can offer a significant performance increase when upgrading an 8600GT with an 8800GTX.

The superior theoretical system bandwidth of the Phenom system does not have a noticeable effect though (see figure 3.5) and its bound for uploading shorts is the same for both series 8 GPUs. The effective bandwidth limitations of the Core2 system can only be seen when uploading unquantized data. For this test there is no benefit from using the 8800GTX, even though the former

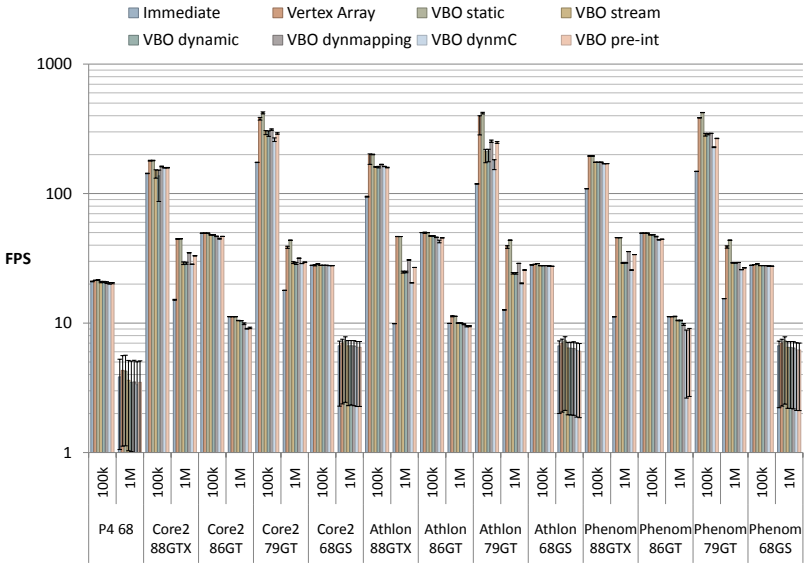


Figure 3.8.: Rendering performance of sphere glyphs on different machines for all available upload modes. The peak performance for the GeForce 7 GPU hints at some overhead or missing optimization for the subsequent hardware generation that is exhibited when the shader load is not very high. DynmC means dynamic mapping including a color attribute and pre-int signifies that data is pre-interleaved in main memory to allow for single-instruction memcopy.

has a higher clock speed and also four times as many stream processors. Altogether the CPU, memory timing, chipset factors and the specific implementation result in a bandwidth utilization of 2.5GB/s (out of 4GB/s, so 63%) when using the 4M float dataset. With shorts the frame rate nearly doubles on the 8800 GTX, so the PCIe utilization is roughly the same, however then the 8600 GT does not have enough computational power to match the performance of the high-end card.

A different approach to reduce the necessary upload bandwidth is the usage of parameter textures. Since in molecular visualization there are only a restricted number of classes – usually atoms/molecules of a certain type/element – all the attributes that are constant per type can be stored in a texture kept in GPU memory and thus require no upload at rendering time at all. Thus, parameters like color and radius are reduced to a single type index that is later used to look up the actual values in the parameter texture.

While this approach should improve performance significantly, practical measurements prove otherwise: using dipole glyphs (see below) which have 5 parameters plus a color per type and thus should allow a significant conservation of bandwidth, the frame rate does not increase significantly overall. The only CPU/GPU combination that benefits from the reduced upload is the Athlon64/8800GTX, which is obviously bottlenecked by the CPU and can only offer the full performance of the GPU (that is, the same frame rates as on the Core2 and Phenom systems) when the load on the CPU side is reduced by looking up the glyph parameters from a texture (see figure 3.9). The figure also shows that parameter textures can be more convenient for simpler shaders on fast GPUs, as reduced fragment load – emulated by reducing the glyph size to 10% – will increase the performance gain offered by the lookup on the 8800GTX significantly.

Comparing the performance of the GeForce 7900 and the 8600 board, it seems that even though the 32 shaders of the newer GPU run at higher clock speeds, they cannot be allocated in such a way to result faster than the 24 pixel shaders of the GeForce 7900. However the older GPU cannot benefit from the reduced fragment load per primitive the way the newer card can, which might be caused by the finer granularity of execution the GeForce 8 series offers (smaller groups of pixels that have to run exactly in parallel).

Other than optimization itself, the main reason why especially the bandwidth requirements of the proposed rendering approach must be as low as possible can be found in the innovation cycle for peripheral interconnects in PCs and Workstations. While graphics chips have perhaps the shortest innovation cycle in the computer industry overall (for various reasons, but most notably the tendency of entertainment companies to propose increasingly realistic graphics for computer games, which in the end is a driving force in the PC market), the overall architecture changes rather slowly. Looking at the evolution of graph-

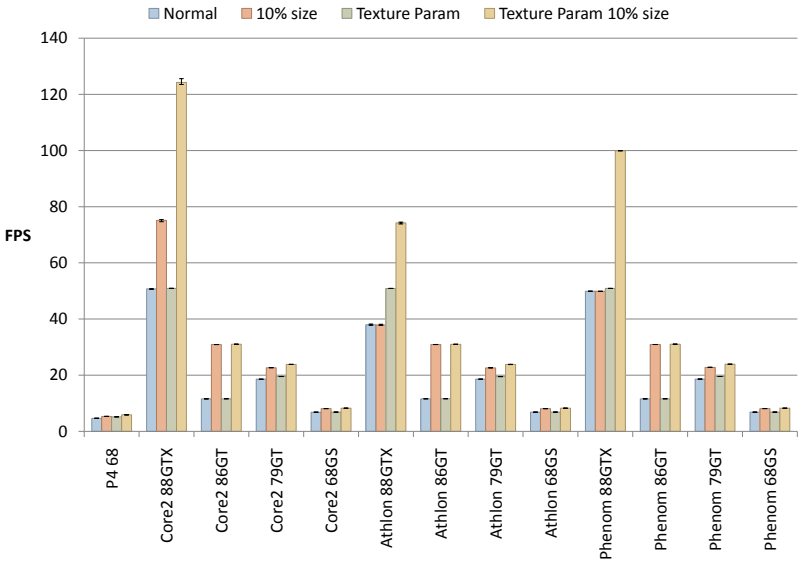


Figure 3.9.: Rendering performance for 500,000 dipole glyphs (see figure 3.13). For testing purposes, the glyphs have also been scaled to ten percent of their size to reduce the fragment load per primitive.

ics card buses and the GPUs themselves (see figure 3.10 and table 3.2), one can see, for example, that bandwidth from CPU to GPU has quadrupled in the same period in which GPU complexity has increased almost fifty-fold. GPU performance obviously is not a direct function of transistor count, but scales even better, however it is difficult to find any benchmark that has been in use for the last ten years and still gives meaningful, comparable results, especially considering that the transistor count not only stems from higher parallelization, but also increasingly complex functionality. The positive effect of the technological advance in GPU technology is that, because of backward compatibility, algorithms running directly on the GPU benefit from the improved performance of new graphics chips automatically and thus will increase in performance, while the available upload bandwidth cannot not be relied upon to increase at a similar rate.

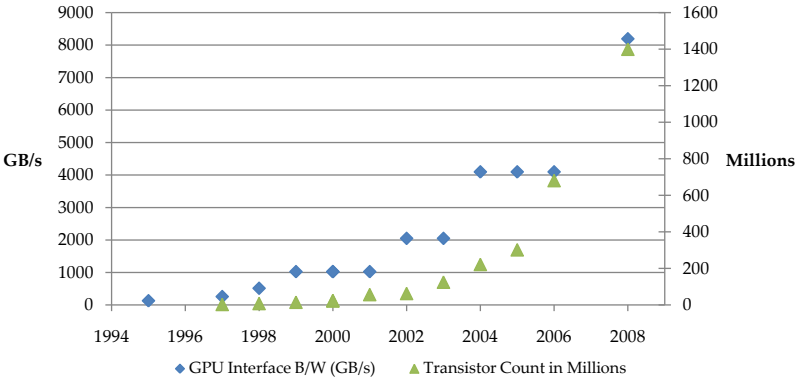


Figure 3.10.: Bandwidth and transistor count increase for the major consumer card generations by Nvidia.

Chip	Name	Release	Slot Type	Bus B/W (MB/s)	Transistors (Millions)
NV1	-	1995	PCI	128	?
NV3	RIVA128	1997	AGP 1x	256	3
NV4	TNT	1998	AGP 2x	512	8
NV5	TNT2	1999	AGP 4x	1024	15
NV10	GeForce	2000	AGP 4x	1024	22
NV15	GeForce 2	2000	AGP 4x	1024	25
NV20	GeForce 3	2001	AGP 4x	1024	57
NV25	GeForce 4	2002	AGP 8x	2048	63
NV30	GeForce FX	2003	AGP 8x	2048	125
NV40	GeForce 6	2004	PCIe x16	4096	222
G70	GeForce 7	2005	PCIe x16	4096	302
G80	GeForce 8	2006	PCIe x16	4096	681
G200	GeForce 200	2008	PCIe2 x16	8192	1,400

Table 3.2.: CPU-GPU Bandwidth and transistor figures for the major consumer card generations by Nvidia.

Spheres

The next stage of the OpenGL pipeline that has to be optimized is the vertex shader. It produces the primitives that are rasterized to the fragments which are going to be processed by the fragment shader. Therefore, a tightly-fitting screen-space bounding geometry for the actual glyph must result from this stage as well as the parameters of the implicit surface. The basic glyph useful for molecular visualization, a sphere, only needs a position, a radius and a color. If the position is quantized, a uniform parameter can be used to set the parent's position and subspace spanned, so the absolute position can be restored in the vertex shader, as has been done in [HE03]. Two different bounding geometries have been tested: an axis-aligned cube and a screen-space rectangle. The cube corresponds to the axis-aligned cube exactly enclosing the sphere, is trivial to compute, but reliably overestimates the required billboard size (at least when a perspective projection is used), a fact that is aggravated by the additional size increase when using single points as billboards: in this case additional fragments are generated, since in OpenGL points always have to be square. To avoid the generation of too many superfluous fragments, a tighter-fitting screen-space rectangle can be used instead.

The tightly-fitting screen-space rectangle is calculated as follows: exploiting the fact that the silhouette of a sphere can be captured using tangent planes perpendicular to the view's *Up* and *Right* vector, the whole problem can be broken down to three nested right triangles for the horizontal and the vertical

screen-space extent each. Figure 3.11 shows the concept by example of the horizontal extent, the other variant being analogous. Euclid's theorems make it easy to determine all the lengths in this diagram:

$$p = \frac{r^2}{|\vec{SC}|}, \qquad q = |\vec{SC}| - p, \qquad h = \sqrt{pq} \tag{3.1}$$

Calculating B_0 is trivial, however $B_{1,2}$ are more costly. The quickest way probably is to project $\vec{SC}$ into the coordinate system spanned by the viewing direction V and right vector R of the current view. Then coordinate inversion yields the correct perpendicular direction (see the dashed segments in the diagram). Using the vectors $\vec{p}$ and $\vec{h}$ from both the horizontal and the vertical tangent planes, the resulting four extents of the billboard are projected into screen space. For the points, the maximum and minimum per screen-space axis are used to determine the respective size.

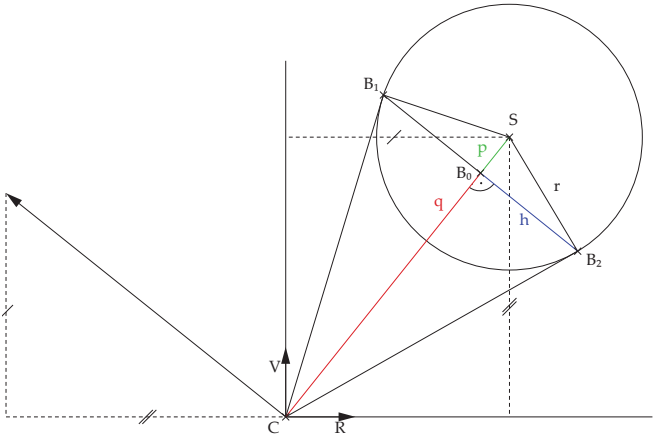


Figure 3.11.: Bounding geometry for spheres based on the silhouette as seen from the 'camera position' C. The tangent planes touch the sphere at the points B_1 and B_2, the vector resulting from the coordinate inversion (dashed lines) shows the direction for the construction of $\vec{h}$.

In theory, once a tightly-fitting bounding geometry is available, the point to be rasterized could be replaced by a quad to reduce the number of 'superfluous' fragments generated, however this not always yields superior results even for cylinders, which usually have a much more inconvenient aspect ratio than a

sphere (see below), so the approach was not implemented in the visualization tool.

The final step to obtain the glyph requires the rendering of a perspective-correct sphere in the fragments rasterized from the vertex shader output. This is achieved through per-fragment raycasting of the implicit sphere surface. First the vector which connects the eye to the current fragment is computed from the fragment's *window position* $WPOS$ as available in the fragment shader stage. x and y are then converted to the *View Coordinate System* (top t, bottom b, left l and right r are the parameters of the viewport, w and h its sizes in pixels)

$$\overrightarrow{s} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}_{VCS} = \begin{pmatrix} \frac{x_{WPOS}}{w} \cdot (r-l) + l \\ \frac{y_{WPOS}}{h} \cdot (t-b) + b \\ -z_N \end{pmatrix} \tag{3.2}$$

According to the OpenGL specification, the z component of the cast ray can be directly set to the near clipping plane distance z_{near}. To reduce the ray intersection calculation as much as possible, the fragment program works in local glyph space, so the sphere is always centered at $(0,0,0)^T$ and the eye position is translated accordingly in the vertex program and passed through as a varying parameter. Thus the computation is only done once per vertex instead of once per fragment. When the sphere is not intersected, the respective fragment is discarded, otherwise it is Phong-shaded. The intersection point can optionally be transformed back into the world coordinate system and then transformed through the built-in OpenGL matrices and mapped to the $[0,1]$ OpenGL depth range, so that the glyphs can be correctly combined with other primitives and native OpenGL polygons.

Cylinders

Cylinders are another handy primitive for molecular visualization, since the ball-and-stick style (see figure 3.2) as well as the stick (or licorice) style make use of them. Cylinders need an additional orientation parameter, which is most conveniently uploaded as a quaternion (4 floats) and converted to the corresponding transformation matrix (9 floats) in the vertex shader. Even without strong perspective distortion, the silhouette of a cylinder in such datasets is usually elongated, resulting in an especially unfavorable ratio of discarded fragments when just rendering square billboards. The rendering performance thus should benefit greatly from a tightly-fitting bounding geometry. The calculation basically works in the same way as for spheres, except that the local coordinate system can be used to obtain an object-aligned bounding rectangle for an even tighter fit.

Three possible approaches for the geometry result: either a single point is uploaded and view-aligned bounds are used, or a quad is uploaded with corner indices for object-aligned bounds such that upload bandwidth is traded for

lower fragment shader load. With current-generation graphics hardware it is additionally possible to expand an uploaded point into a triangle strip in the geometry shader stage to exploit the benefits of both the single-point upload and the tighter-fitting quad. It should be noted that the geometry shader has to output a much higher number of attributes per vertex than in comparable approaches (e.g. the billboard generation in [LVRH07]), thus putting a significant load on the GPU. The proposed approach still needs a transformed camera and light position passed to the fragment shader in addition to the glyph parameters. Unfortunately, current Nvidia GPUs are extremely sensitive to the total number of attributes emitted, resulting in unsatisfying performance. Figure 3.12 demonstrates that only the Athlon machine benefits from the bandwidth reduction, and also only when a GeForce 8800GTX is employed since the mid-range version does not have enough resources to make the additional geometry shading cost affordable. For other hardware combinations the brute-force approach uploading quads is usually more convenient to use. It can be deduced from the point primitive measurements that with current hardware, the fragment shader stage is so thoroughly optimized that a 50% overhead in fragments that will be killed anyway (for a cylinder with aspect ratio 2:1 as the ones used in the measurements) can be put up with when using only moderately complex shaders, so the 'suboptimal' point primitives are still a reasonable approach. At least for this specific case the geometry shader is too expensive to even justify an upload bandwidth reduction by 75%.

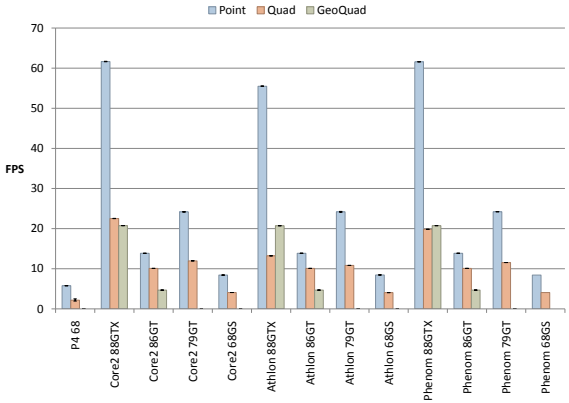


Figure 3.12.: Cylinder primitive performance depending on the bounding geometry generation. Only on very slow machines the bandwidth conservation offered by the geometry shader improves the framerate.

The cylinder glyph is rendered in its local coordinate system (same as the sphere), so the orientation is mapped to orbiting the eye point around the glyph. The final rotation is obtained by combining the camera orientation quaternion and the glyph orientation quaternion. The respective multiplication is done in the vertex program as well as the subsequent transformation into the corresponding rotation matrix. This matrix also has to be passed to the fragment shader in order to correctly orient the per-fragment ray and transform the intersection point back to world space, so that an OpenGL-conforming depth can be calculated from it. If the cylinder is used by itself, it can be drawn normally, which in contrast to a sphere means that both intersections are needed since it is possible to look inside the cylinder. Alternatively the cylinder can be capped (by itself or other primitives), making the far intersection superfluous. This is the case, for example, when a cylinder is used in a ball-and-stick visualization where the ends of the cylinders are hidden anyway.

Dipoles

In addition to the rather generic basic primitives, for the application of molecular dynamics used for nucleation simulations, a specifically tailored glyph was needed. The molecules present in these simulations consist of several atoms, but since at the ITT a Lennard-Jones potential is used for the simulation itself, it is desirable that the molecules be rendered accordingly. Consequently each molecule consists of one or two Lennard-Jones centers and can be polarity-free, di-polar, or quadru-polar. The classical glyph employed in thermodynamics is a pair of spheres with the respective van-der-Waals radius of the atoms. The difference in radius can make it easy to distinguish which end of the molecule is which, however the polarity has to be known by the user, which should not be necessary as it burdens the user with a task of the visualization itself (to convey information [HJ05]), that is showing the data so the user can intuitively understand it. In collaboration with our thermodynamics colleagues the following solution was devised: since school a significant portion of people should be aware of magnets as used in experiments in physics, for example, which often are conveniently marked with green and red ends to signify the polarity. As an emulation of these markings, a cylinder can be rendered along the axis of a two-center Lennard-Jones molecule with the same red-green coloring for easy recognition. The addition of this magnet metaphor also allows the mapping of the charge to the length of the cylinder, as well as of an additional property to its radius, effectively allowing for 5 attributes (3 radii, the Lennard-Jones centers distance, and the cylinder length) to be displayed alongside position, orientation, and color (see figure 3.13). The resulting glyph is sufficiently simple to be generated on the fly from implicit surfaces in the fragment program, reducing the uploaded data to only one position and orientation plus 5

parameters (12 floats) instead of three positions plus the cylinder orientation and 4 parameters (17 floats). This corresponds to a reduction of 30% in upload bandwidth consumption.

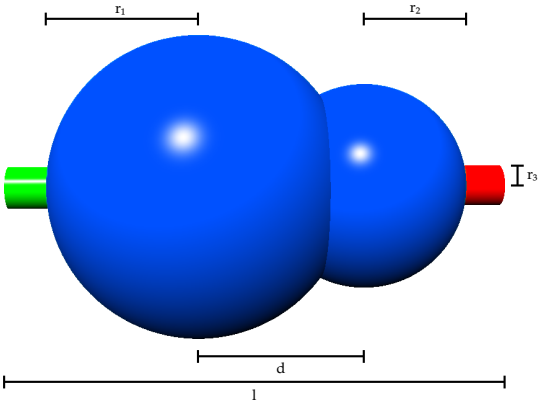


Figure 3.13.: Dipole glyph and its adjustable parameters.

Assuming that the dipole glyph is always at the origin of a local coordinate system (same as the other glyphs), with the cylinder lying along the x axis, the implicit surface generated from these parameters is defined as

$$\left(\begin{pmatrix} x-\frac{d}{2} \\ y \\ z \end{pmatrix}^2 - r_1^2\right) \cdot \left(\begin{pmatrix} x+\frac{d}{2} \\ y \\ z \end{pmatrix}^2 - r_2^2\right) \cdot \left(\begin{pmatrix} 0 \\ y \\ z \end{pmatrix}^2 - r_3^2\right) = 0 \tag{3.3}$$

Additionally, the infinite cylinder must be trimmed and capped:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \cdot \overrightarrow{x} = r_1 + \frac{l}{2}, \quad \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} \cdot \overrightarrow{x} = -r_2 - \frac{l}{2}. \tag{3.4}$$

Intersecting these surfaces with a generic ray cast from the eye position (see also the fragment position calculation in (3.2))

$$\overrightarrow{r} = \lambda \cdot \overrightarrow{s} + \overrightarrow{pos_{eye}} \tag{3.5}$$

yields 3 quadratic equations. The local coordinate system and orientation simplification is analogous to the cylinder primitive, however this glyph already has so many parameters that need to be passed from the vertex to the fragment

shader, that the varying parameters available in hardware at the time of publication are all expended when the orientation matrix is passed to the fragment shader as well.

The next step consists of solving the three quadratic equations given by intersecting the ray with the two spheres and the cylinder. First the three radicands are computed for an 'early' discarding of the fragment if all three of them are negative (and no surface is hit). The combined sphere hit result is also kept for later use. After calculating the roots, illegal results are discarded by assigning a ray parameter λ which positions the fragment behind the far clipping plane. The cylinder is a bit tricky to calculate.

For the cylinder, the near and far intersections are needed (C_N, C_F). Then the planes forming the caps cutting off the cylinder according to the length l must be calculated and the respective λ is used to decide which intersection is on the near plane (P_N) and which is on the far plane (P_F). Now the five different cases describing which part of the geometry is going to be hit can be distinguished (see also figure 3.14).

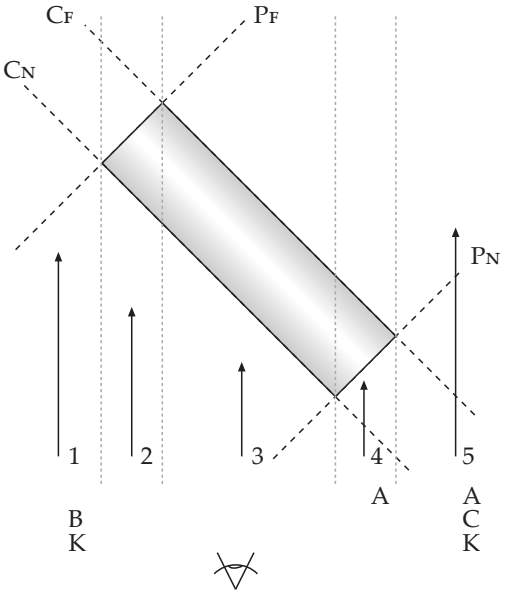


Figure 3.14.: The different cases when raycasting the cylinder alongside indications where the distinguishing conditions (namely A, B, C, K) are true

The conditions of the five possible cases are the following, listed along with the action that needs to be performed:

$$\begin{array}{ll} 1: \lambda_{C_F} > \lambda_{C_N} > \lambda_{P_F} > \lambda_{P_N} & Sphere/Kill \\ 2: \lambda_{C_F} > \lambda_{P_F} > \lambda_{C_N} > \lambda_{P_N} & Cylinder \\ 3: \lambda_{P_F} > \lambda_{C_F} > \lambda_{C_N} > \lambda_{P_N} & Cylinder \\ 4: \lambda_{P_F} > \lambda_{C_F} > \lambda_{P_N} > \lambda_{C_N} & Cap \\ 5: \lambda_{P_F} > \lambda_{P_N} > \lambda_{C_F} > \lambda_{C_N} & Sphere/Kill \end{array} \tag{3.6}$$

These conditions can be condensed into distinguishing conditions, thus it is sufficient to use the following three conditions for deciding the rendering result:

$$A: \lambda_{P_N} > \lambda_{C_N}\ , \quad B: \lambda_{C_N} > \lambda_{P_F}\ , \quad C: \lambda_{P_N} > \lambda_{C_F} \tag{3.7}$$

The condition $K = B$ or C discards all cylinder information by again setting an impossibly high ray parameter λ, so that either the spheres are rendered (because the corresponding λ is smaller) or the fragment is discarded if there was no sphere hit in the first place. Based on the results of these calculations it can also be decided which of the available normals is the correct one:

$$\overrightarrow{N_{cap}} = A \cdot \bar{C} \cdot \overrightarrow{N_{cap}}\ , \quad \overrightarrow{N_{cyl}} = \bar{A} \cdot \bar{B} \cdot \overrightarrow{N_{cyl}} \tag{3.8}$$

The discarding condition K conveniently also ensures $\bar{B}$ and $\bar{C}$, so the only remaining condition is A (the cap is hit) and $\bar{A}$ (not the cap, but the cylinder is hit). The only case that is neglected by the above is looking through the cylinder, thus having huge λ for the cylinder hits (or even infinity). This is however already corrected by the discrediting of negative radicands and will not come to pass. The resulting surface is then Phong-shaded. The color for the poles of the cylinder is decided as a by-product using the sign of the cylinder intersection x coordinate.

Results and performance

Applying these concepts to the visualization of molecular dynamics datasets simulated at the ITT was quite a success. Researchers were used to classical polygon-based visualization with a rather simple in-house tool, whose performance could be bested by more than two orders of magnitude with (at the time) current GPUs. Previously, it was actually impossible to view datasets with more than 10.000 molecules at all. The additional increase in quality was judged very positively. Experimenting with larger datasets, however, made it also clear that the perception of depth was not convincing despite the rather strong perspective parameters. Additional depth cues can improve this situation [WE02], so a simple scheme that attenuates the lighting depending on the distance from the viewer was added. The difference between the two modes can be observed in figure 3.15. More recently, better, though still interactive,

cues for the depth complexity have been proposed, for example using ambient occlusion, an interactive radiosity approximation [TCM06].

The rendering performance for 500,000 different glyphs can be seen in figure 3.16, evidently decreasing depending on their individual complexity. Figure 3.8 already showed, however, that performance does not decrease linearly with the number of rendered glyphs, even though it directly depends on the number of fragments generated. This can easily be explained by the nature of the proposed hybrid object/image-order approach: The more molecules are rendered, the less screen space each one occupies, reducing the number of fragments to process significantly, which compensates partly the performance impact of many glyphs. When zooming in, many fragments are even clipped since they come to lie outside of the current viewport, which makes up for the visible glyphs that become very large. The achievable speedup per graphics card generation can also be observed here, in accordance to the prognosis already made in section 3.2.2.

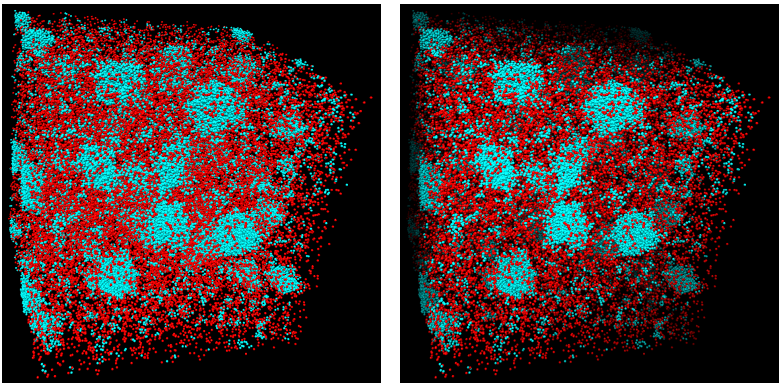


Figure 3.15.: CO_2 nucleation simulation with 65K items and no depth cues (left image). The structure (and Z-order) of the dataset is much easier to perceive when a simple depth cueing by light attenuation is used (right image).

Clusters Visualization

When studying the nucleation in gases, the improved visualization can already help researchers, however the datasets are still heavily cluttered if the simulated gas is strongly oversaturated. To better distinguish the clusters from the remaining monomers, an abstract representation was included. Since in realistic (in terms of saturation) simulations the clusters in the end are just very

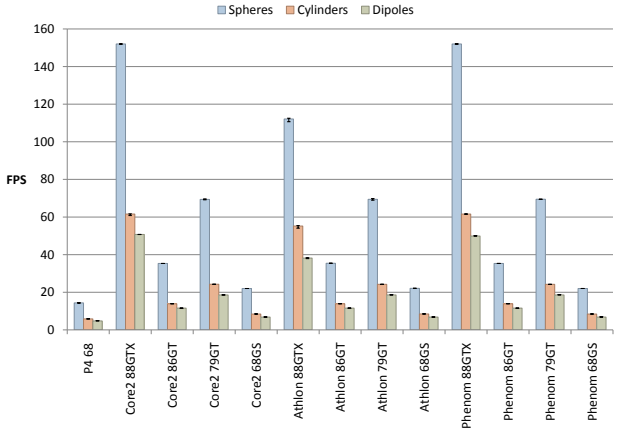


Figure 3.16.: Glyph rendering performance for different glyph/hardware combinations.

small droplets, it can obviously be expected from them to have, by and large, near-spherical shape as soon as they reach a sufficiently large size (induced by the surface tension) . This led to the decision to integrate the ellipsoid shader by Thomas Klein [KE04] as a representative of clusters. To improve the rendering strategy, a shallow hierarchy was added that basically contains clusters and the respective information for rendering them (semi-axes lengths) as inner nodes and the contained molecules as leaf nodes, plus an additional node for all the remaining monomers. Using shaded ellipsoids, the user can perceive the position and extent of the molecular clusters more easily, even if a large number of monomers occlude portions of the ellipsoid as shown in figure 3.17. The ellipsoids can also be shown as a transparent overlay over the monomers contained in a cluster (see figure 3.24, right image). The ellipsoid parameters are calculated similarly to [SGEK97]: the position $e_j(t) = (\bar{x}(t), \bar{y}(t), \bar{z}(t))^T$ of the center of the ellipsoid is the average position of all molecules forming the cluster. The main axes are determined using the eigenvectors of the covariance matrix, which is then sorted according to the eigenvalues. The normalized eigenvector of the largest eigenvalue is used as first main axis, while the other two main axes are then calculated using the eigenvector of the second largest eigenvalue and cross products. The radii of the ellipsoid are finally determined by projecting the contained molecule positions onto the resulting main axes. Since this method actually describes the bounding cuboid, the resulting ellipsoid will not wholly enclose some of the border molecules, but this number is negligible since a droplet can be safely assumed to have roughly spherical shape, as mentioned earlier.

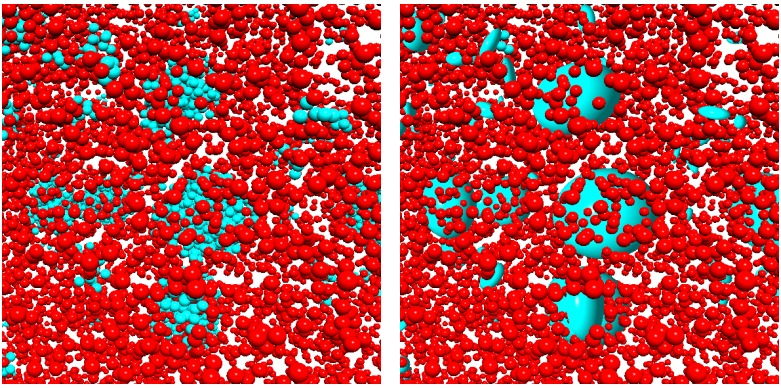


Figure 3.17.: Molecular clusters are normally represented using a simple color coding (left). Using ellipsoids allows to more easily perceive the shape of the clusters (right).

3.2.3. Time-based Data

The algorithms proposed so far form the building blocks for a versatile static visualization, however, to help with the analysis of simulations, the point cloud renderer needed an improved data handling to make the efficient rendering of time-based datasets possible. Even more important than the basic playback of simulated time steps are the methodical advantages that can result from the use of a time-based visualization. In this case, the verification of the correctness of a simulation run and, for thermodynamics applications, the reliability of the employed clustering criterion are two important aspects. For the calculation of the nucleation rate and the critical cluster size, it is essential to make use of a meaningful definition for molecular clusters because only if the clusters are correctly detected, the corresponding metrics will yield correct results. The verification of the clustering results with the aid of interactive visualization is as important as the comparison of the resulting nucleation rates with experimentally acquired values. By observing the detected clusters and their interaction with the surrounding monomers, problems (and bugs) in the employed molecule cluster detection algorithm can be isolated. An additional question that arises is also how monomers interacting with several clusters in a short time have to be interpreted: sometimes they might indicate a cluster merging with another, sometimes two erroneously separated clusters.

The datasets mainly used in screen shots and for performance measurements in this work all result from molecular dynamics simulations conducted at the ITT. A typical simulation spans a fraction between 0.2 and 0.5 of a nanosecond and contains tens to hundreds of thousands of molecules. The simula-

tion time step for these simulations usually is 1fs in length. That frequency is much too high for visualization, because on the one hand, the storage requirements for the resulting 200,000 to 500,000 configurations are too high (hundreds of GB) and on the other hand the temperature in the simulation domain is low enough that the molecules move so little per time step that the playback would take far too much time without offering additional insight. The empirically determined temporal subsampling of 100 still leaves enough data such that straightforward linear interpolation between time steps does not produce any artifacts like molecules moving on visibly piecewise linearized trajectories. Thus time-based visualization of the single molecules or atoms basically consists of loading two configurations and interpolating the respective positions between them. The molecule color is either mapped from a table based on its type, or set to a color reserved for the molecules being part of a cluster (see figure 3.17).

The molecule clusters themselves have several attributes that require special consideration. Three different mappings are available for the cluster color: the default coloring of the ellipsoid corresponds to the user-defined cluster membership color also used when the molecules are rendered individually to obtain visual coherence. There are two alternative coloring schemes available: the first one uses a palette of clearly distinguishable colors from which a color is selected using the ID of the cluster modulo the number of entries in this color table. This color coding allows to easily check the tracking of clusters over time when the monomers are filtered out. The resulting rendering is very taxing as it causes high sensory load for the user, so it is mainly used for debugging purposes. The second alternative coloring scheme emphasizes the evolution of the molecular cluster. Three user-defined colors represent the major evolution tendencies of the clusters. The default colors are light yellow for clusters keeping their size, light green for growing and light red for shrinking clusters. These colors are interpolated according to the actual evolution (e.g. slow-growing clusters are colored in light greenish yellow; see figure 3.18).

Interpolating the position and orientation of two ellipsoids is not as straightforward. First of all, an ellipsoid is symmetric, so two axes can be swapped without the shape changing. Second, and this is the more critical aspect, interpolation can either minimize the scaling or minimize the rotation of the ellipsoid. The first approach will interpolate between the radii of the ellipsoid, which results in the biggest radius remaining the biggest radius. The orientation quaternions are then directly interpolated using *SLERP* [DKL98]. The second approach will select pairs of axes such that the rotation is minimized. The orientation quaternion of the targeted configuration must then be recalculated. This computation can be performed at loading time. The differences between these two interpolation methods are only visible if very unstable clusters are used, and then both approaches result in rather particular animations, either with spinning ellipsoids or wobbling ellipsoids. Since these effects can

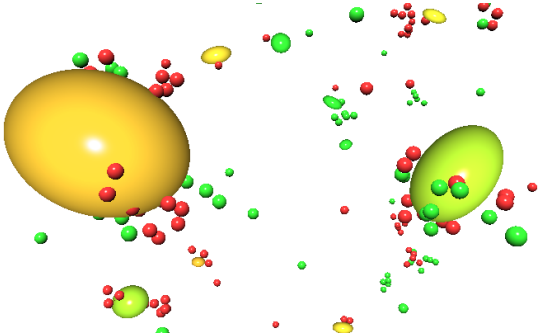


Figure 3.18.: Molecular cluster ellipsoids with color coded evolution and filtered monomers, color-coded whether they are joining or leaving clusters. Clusters with colors between yellow and red are shrinking (left cluster), while clusters with colors between yellow and green are growing (right cluster).

be observed only rarely, the direct interpolation of the quaternions without any recalculation of these values was opted for. The spinning clusters can then be used to determine whether a dataset is highly unstable, while wobbling could be misinterpreted as reflecting the actual situation, leading to a higher perceived correctness of the simulation results than is justified, at least according to the experience at the ITT.

To avoid popping artifacts for monomers and clusters, both are scaled smoothly over time so that monomers have an interpolated size between 100% in the last configuration they are unconnected to a cluster and 0% in the configuration where they first join a cluster, and clusters reach 100% size during playback of the first configuration where they start existing.

Molecule Flow Visualization

The correct detection of a cluster over time implies that it exhibits a certain stability. It is not impossible that a once agglomerated molecule leaves such a droplet again, but it needs additional energy to overcome the inter-molecular attraction as per the van-der-Waals forces (and additional ones). If it seems that lots of molecules leave a cluster all the time, however, the suspicion should arise that the cluster potentially is not detected correctly in the first place. It is even worse if such molecules join in a secondary cluster which might in turn dissolve as well, with the molecules dropping back into the original cluster. This can be a sign that the two clusters should have been detected as one. Hence the idea of tracking the molecule exchange between clusters, which

should allow the researchers to detect clusters that 'feed' smaller clusters and vice versa. Related work for such a flow visualization is manyfold: it has been tackled recently with mostly texture-based, dense representations. The texture-based approach was introduced with LIC [CL93] and has seen many improvements and hardware-accelerated implementations [JEH02] since, even on 3D surfaces [WE04] and for 3D fields [WSE05, IG98, SFCN02]. Since an integral part of the information that needs to be visualized consists of the droplets themselves, a dense representation for the surrounding space could not meet expectations as it would clutter the rendering too much. Therefore a simple, sparse visualization, more closely related to streamballs [BHR+94] and other glyph-based flow visualization techniques [RSBE01] was chosen.

The first step towards generating such a glyph-based flow visualization is a filtering of the monomers according to their importance for the evolution of a molecular cluster. All the others should sensibly not be included in the inter-cluster flow visualization. As basis for this filtering the *cluster time distance* was devised. The groundwork for this are the following definitions for the elements in a dataset: $\mathbf{T} = \{0, ..., t_{max}\} \subset \mathbb{N}_0$ is the time line of the dataset with the discrete configuration times $t_i \in \mathbf{T}$, and $\mathbf{M}$ is the set of all molecules in the dataset with the trajectory of molecule $m_i \in \mathbf{M} : t \mapsto (x_i(t), y_i(t), z_i(t))^T$. Molecular clusters are identified by a natural number. The value zero is used to represent molecules that are not part of a cluster, hence the cluster membership of a molecule m_i can be defined as:

$$c : \mathbf{M} \times \mathbf{T} \rightarrow \mathbb{N}_0, (m_i, t) \mapsto c(m_i, t) \tag{3.9}$$

A molecular cluster can then be defined as the time-dependent set of the contained molecules:

$$\mathbf{S}_j(t) := \{m_i \in \mathbf{M} | c(m_i, t) = j \wedge j \neq 0\} \tag{3.10}$$

Finally, the *cluster time distance* is a signed value for each molecule, representing the distance in time to the next cluster:

$$ctd(m_i, t) = \begin{cases} t_c - t & \text{if } \exists t_c \forall t_x : |t_c - t| \leq |t_x - t| \\ & \text{with } t_c, t_x \in \{t_j \in \mathbf{T} | c(m_i, t_j) \neq 0\} \\ nan & \text{else} \end{cases} \tag{3.11}$$

The symbolic constant *nan* is used to indicate that a molecule is never part of a cluster and that the set used in the first case therefore would be empty.

Using this metric, only monomers are rendered whose absolute value of the cluster time distance is smaller than a user-defined threshold:

$$ctd(m_i, t) < \epsilon_{filter} \tag{3.12}$$

To further emphasize the contribution of the monomers to the evolution of molecular clusters, the molecules are color-coded according to the sign of their cluster time distance, showing molecules joining a cluster in green, and molecules leaving a cluster in red, see figure 3.18. One interesting situation is given by monomers starting as red, leaving clusters and then switching their color to green. If this occurs with rather small filtering thresholds, it can indicate cluster detection instabilities.

It can be easily understood that it is hard to perceive these situations, because the changes of the monomers' colors happen rather quickly and in several places in the dataset at once. To overcome this problem, the information of the filtered monomers can be used to produce a pathline visualization by simply connecting the positions in the temporal neighborhood (also used in [KBH04], for example) of these molecules. The pathline information can be generated at loading time, so no additional storage in the input file format is needed. However, the creation of this information increases the memory needed and the workload on the CPU. Using these pathlines allows the user to track the movement of the monomers over a period of time even when the simulation playback is paused, which clarifies the joining, leaving, and rejoining phenomenon described above. However, since the pathlines represent information of several configurations, despite the rather sparse visualization itself, the images tend to be even more cluttered when compared to rendering the filtered monomers. It can nevertheless give a quick overall impression on the clustering behavior of the dataset in general, as a distinctly greenish picture would mean that the clusters are steadily growing, while a reddish picture means that many clusters are collapsing. The latter output should also alert the user, since even very unstable situations at worst yield an even mix of green and red. A simulation with a prevalently collapsing behavior has not yet been encountered, but it would be easy to obtain by placing a single drop in empty space at a relatively high temperature. It is however quite hard to imagine the insight generated by such a simulation.

To further improve the visualization, one possibility would include the aggregation of several pathlines into some kind of pathtube, which however would have to span all the positions of the contained molecules and would in most cases use even more space to display. As an abstraction of this concept, flow groups were devised. A flow group is defined as a set of molecules $\mathbf{M}_f$ moving together from one cluster to another:

$$\begin{aligned}\mathbf{M}_f(e, v, t_e, t_v) := &\{m \in \mathbf{M} | c(m, t_e) = e \wedge c(m, t_v) = v \\ &\wedge \forall t \in (t_e, t_v) : c(m, t) = 0\} \\ &\text{with } e \neq v \text{ , } t_e < t_v\end{aligned} \tag{3.13}$$

So the flow group is defined by the configuration time t_e of its emergence, the ID e of the cluster it is leaving, the time t_v of its disappearance, and the

ID v of the cluster it is joining. This concept was developed in cooperation with our thermodynamics partners especially to investigate the interchange of molecules between clusters. The flow groups are visualized as GPU-generated arrow glyphs (based on the cylinder glyph from section 3.2.2), moving from the averaged position of all molecules in the starting configuration to the averaged position in the ending configuration. All intermediate positions are discarded and replaced by linear interpolation from start to end, so only the information that a molecule exchange has happened remains.

The size of each arrow is calculated from the number of contained molecules. For the mapping from the number of molecules to the size of the arrow the cubic root with the factor provided by the Kepler conjecture ($\frac{\pi}{\sqrt{18}} \simeq 0.74048$) is used, which corresponds to the closest packing of similar-sized spheres:

$$r(\mathbf{M}_f) = r_m \sqrt[3]{\frac{\sqrt{18}}{\pi} |\mathbf{M}_f|} \tag{3.14}$$

with r_m being the radius of the molecules and $r(\mathbf{M}_f)$ being the radius of a flow-group-enclosing sphere $\mathbf{M}_f$. The arrowhead radius, head and tail lengths are then set to the radius of this sphere to obtain a visually similar impression of volume, while additionally emphasizing its flow direction. This approach creates glyphs with a good representation of the amount of molecules in each flow group and allows for a qualitative comparison of flow groups and molecular clusters. To create a further coupling between the size of the flow group and the size of the molecular clusters involved, the flow group's arrow is color-coded. The color is interpolated between two base colors depending on the ratio of the flow group molecule count and the molecular cluster molecule count. A completely blue flow group holds all molecules which formerly formed the molecular cluster, while an almost red flow group only holds very few molecules compared to the number of molecules in the cluster. So the effect that molecular clusters are present but missed by the cluster detection algorithm appears as a cluster vanishing, a rather blue flow group emerging, and a new cluster emerging where the still blue flow group vanishes (see figure 3.19).

This approach is applied to evaluate the stability of different molecular clustering criteria (see section 3.2.4).

Preprocessing and Data Streaming

To support the time-based visualization and the rendering modes for clustering analysis, a proprietary binary file format was developed (see [Gro05] for details of the first version), which already has performance advantages compared to the plain text used at the ITT. This format also supports the different quantization variants introduced in section 3.2.2. The conversion takes place in

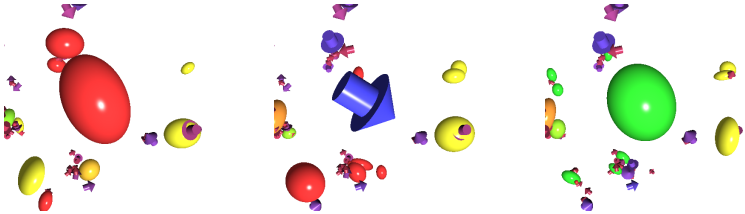


Figure 3.19.: Evaporating molecular cluster (red cluster ellipsoid in the middle of the left image) leaving a flow group (blue arrow, center image) containing almost all molecules of the evaporated cluster, moving slowly and forming a new cluster (green ellipsoid in the right image). The smaller clusters in the left part of the images are surrounded by flow groups, but these are red and the clusters are rather stable.

a preprocessing step, which also helps to compute the additional information that is required for cluster rendering and cluster analysis. The most expensive metadata currently is the cluster time distance, which in the worst case requires information about the whole simulation run for every monomer and thus cannot be calculated on the fly. The flow groups, as well as their contained molecules, are stored separately. Regardless of quantization, a minimalist hierarchy is used to categorize the objects contained in the resulting scene, with one subtree containing the monomers, and one containing the clusters and the associated molecules (see figure 3.20). For ball-and-stick visualizations, there would be an additional subtree containing the bonds, i.e. cylinders. Molecular clusters are represented by inner nodes of this hierarchy, which store all information about the ellipsoidal shape of the molecular cluster. The positions can be stored directly, or they can be placed in a positional hierarchy using relative quantized coordinates as presented in [HE03]. Thus all primitives of one type can be rendered in one go and expensive shader switching is minimized. A linear memory layout of the molecule data in the leaf nodes is created to exploit the advantages of *Vertex Arrays*; all of the hierarchical nodes only store a start and end index into this linear data structure. The use of a preprocessor also makes the support for different input file formats more convenient, as the visualization component only has to support its native, optimized format. The preprocessor has been implemented as a separate program which currently generates the molecular cluster ellipsoids, flow groups, the cluster time distance, and file offsets for seeking within the trajectory. Although each dataset has the same number of molecules in every configuration, these file offset tables are necessary for fast seeking because the sizes of the configurations in one file are not constant due to the varying number of clusters and flow groups per configuration.

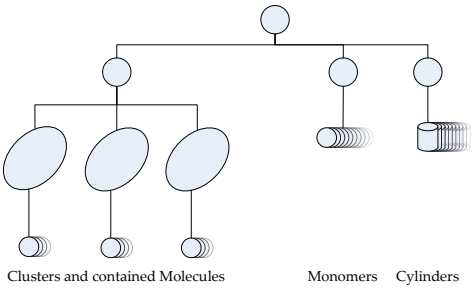


Figure 3.20.: Category hierarchy of the proprietary file format. Leaf nodes are always groups with linear memory layout. The single objects can also be held in a finer hierarchy to allow for the use of quantized relative coordinates.

The use of a hierarchy, however, causes problems when interpolating the positions of the molecules. 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. When considering cyclic boundary conditions resulting in molecules traversing the whole simulation domain between two consecutive configurations, it becomes obvious that a single hierarchy with quantized positions is not applicable to the datasets in question. Since cluster membership is also encoded into the hierarchy, the streaming mechanism must cope with changing parents for all molecules. Hence, one distinct hierarchy per configuration is used and the coordinates of two consecutive configurations are adjusted at loading time. The resulting in-memory configurations no longer represent the simulation state at a discrete point in time, but the simulation state at the beginning and end of a time slice with the length of the simulation time step. Since the hierarchy of the first configuration is used, the second configuration is dequantized and stored relative to the hierarchy of the first one to avoid major precision loss.

The large file sizes of the datasets are the reason why many visualization tools mentioned in Section 3.2.1 fail to load the datasets and to render the molecules at interactive frame rates. Therefore, an *out-of-core* data streaming mechanism was implemented to overcome this problem. First the memory footprint of a single configuration of the dataset is estimated. Because only the number of molecular clusters and flow groups change over time, an approximated value can be determined by examining the first and the last configuration of the dataset. Considering this maximum memory footprint and the amount of memory available, several buffers are created, used to store the current configuration

and to prefetch configurations which will be needed next when the trajectory is played back as animation, controlled by a priority queue and processed by a second thread.

3.2.4. Interactive Cluster Analysis

The presented visualization modes work well in practice, however they only display qualitative information about the dataset and thus are not sufficient by themselves to help with the verification of clustering algorithms. It is easy to identify candidates for failure or further investigation, but if a cluster does not vanish all at once and loses only part of its members, it is difficult to see whether all of them will rejoin it afterwards or just a few. Using additional tools (graph-drawing software or statistical tools, which are widely available) for the quantitative analysis complicates the workflow since neither synchronized selection nor coordinated views are available. The user would have to search himself for low-level information like the IDs of the molecules or clusters and match them manually to the visualization. Despite the visualization tool being mainly as generic as possible, for this specific problem the molecule view was complemented with a schematic graph view representing information about the evolution of the detected molecular clusters over time as well as the molecule flow between these clusters (see figure 3.21). This graph is similar to the visualization proposed in [LBM+06] for the analysis of fluid mixing. In the schematic view the flow is visualized as lines connecting the different molecule clusters, which helps tracking the potential classification problems over time. To reduce clutter when displaying the molecule flow, the selection is synchronized in both views (a technique also known as *brushing & linking* [Kei02]) and can be used to filter the graph display.

The two object classes represented in the schematic view are the molecule clusters and the flow groups. The horizontal axis of the view represents the time line $\mathbf{T}$ of the dataset. Molecule clusters $\mathbf{S}_j(t)$ are represented as horizontal lines. The size $|\mathbf{S}_j(t)|$ of a molecular cluster can be encoded as the thickness of the line. Because this is a time-varying value, the thickness changes when moving along the line horizontally, so that the line appears as a symmetric stripe.

The flow groups are represented as Bézier curves connecting the molecular cluster lines e and v at the times t_e and t_v, encoding their size $|\mathbf{M}_f(e, v, t_e, t_v)|$ as thickness as well. Thus 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 leaving as one flow group. To ease the tracking of the flow groups, different color mappings are supported, from a defined solid color with simple alpha blending to coloring based on the cluster IDs e and v to visually cluster the different flow groups between pairs of clusters. The palette is restricted by default to eight colors so their hue can be easily distinguished.

By default, these cluster lines are vertically ordered by their IDs j, resulting in a

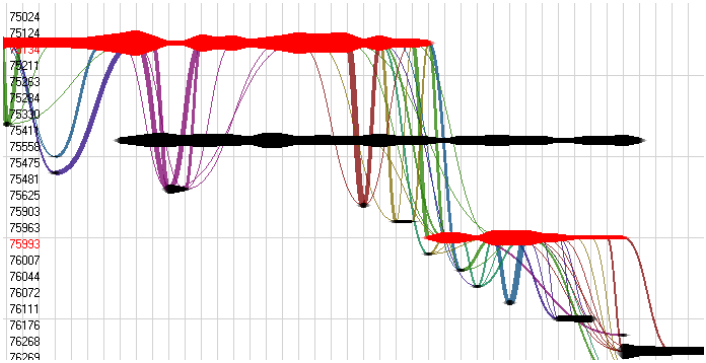


Figure 3.21.: The schematic view of the molecular cluster evolution with two selected clusters (red) and all corresponding flow groups connecting them to other clusters (black) shows the 10,000 methane nucleation dataset with geometrical cluster detection. The vertical axis displays the respective cluster IDs. The flow groups' colors are based on the IDs e and v.

slope-like visualization (as clusters are numbered in the order of their discovery) but can also be rearranged by the user to generate clearer images. To automatically improve the layout, the canonical approach would be to put clusters with connections (flow groups) closer together, possibly even the closer the more molecules are contained in the flow group. The quickest way to derive an ordering would have been to make use of the FastMap algorithm (see chapter 4) and project the different clusters into 1D, effectively yielding the vertical sorting in the schematic view. However, since the flow groups form a sparse graph, the correspondent distance/similarity matrix is sparse as well, thus making it very difficult to choose pivots that have a defined distance to all other clusters, and resulting in a quite unsatisfactory layout. One of the common approaches for graph layout, the force-directed layout [BETT98] is much better suited for this problem. This method is quickly implemented but very expensive ($O(N^3)$), however the number of selected clusters and the ones connected to them is very low in practice (usually about 30 in the datasets that were tested), so this does not represent a problem. The employed forces are based on the Coulomb repulsion

$$F_r = k_c \frac{|q_1||q_2|}{r^2} \tag{3.15}$$

(where k_c represents the electrostatic constant and q_1 and q_2 the charges), and Hooke attraction

$$F_a = -s\delta l \tag{3.16}$$

where s is a spring constant, and δl the difference from the spring's rest length. Setting a damping of 10% and leaving all the constants/charges as well as the rest length at 1 results in a layout with sufficiently localized connections (see figure 3.22). For this layout, F_a has also been scaled with the flow group size to quickly pull clusters near the ones they have a high rate of molecule exchange with.

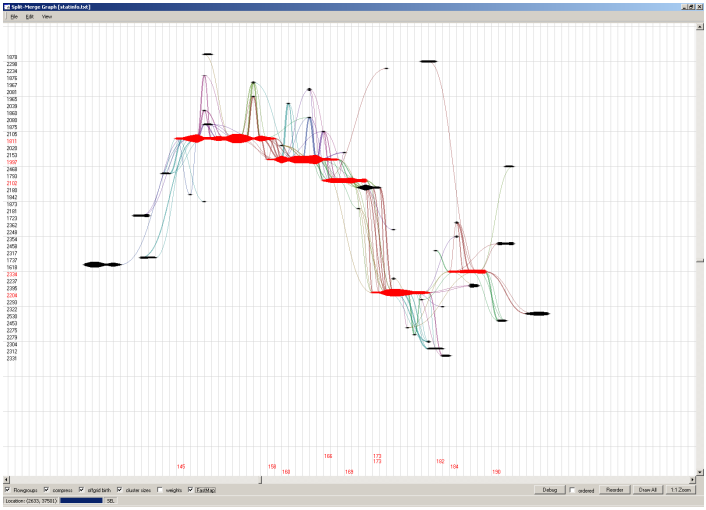


Figure 3.22.: Force-directed layout of cluster evolution data to shorten especially those connections representing large flow groups, thus making the molecule exchange easier to follow.

The schematic view can be interactively zoomed and panned by using the right mouse button and the mouse wheel to ease a quick exploration of the dataset. Molecular cluster lines can be interactively selected by either clicking on the line in the schematic view or by picking the corresponding molecular cluster ellipsoid in the molecule rendering. Selected clusters are rendered with a different color in the schematic view and rendered with a halo for better perceptibility in the molecule view[2]. The schematic view can then be filtered to only show the selected cluster lines, the connected flow groups, and the cluster lines directly connected to the shown flow groups (no further transitivity).

[2]This idea was contributed by one of the reviewers for VIS'07. I would like to thank him/her for this suggestion.

Performance

This combined solution has been tested with four datasets: two datasets of methane nucleation simulations, one with 10,000 spherical molecules and a trajectory of 5000 configurations, and one with 50,000 spherical molecules and 10,000 configurations. The R-152a dataset represents a nucleation simulation of Difluoroethane with 100,000 two-center molecules and 1000 configurations. These three datasets are used to demonstrate the different visualization modes and their performance, including the molecular cluster ellipsoids and the flow groups. The last dataset stems from a biochemical simulation of TEM-1 β-lactamase in water solvent with 28,000 atoms (spheres) and 2000 configurations, and is mainly used for comparing the system performance for time-based data to existing applications like VMD.

Table 3.3.: Performance and memory footprints for different datasets. *Fps* is an average value calculated over the complete trajectory. *Reloading* specifies the number of configurations consistently loadable per second.

dataset	memory footprint	fps	reloading
10,000 methane	1.561 MB	350	43
50,000 methane	7.801 MB	127	29
R-152a	15.600 MB	53	7
Tem1	1.114 MB	382	64

VMD or Chimera, although widely used, are implemented quite inefficiently especially for time-dependent datasets. The TEM-1 dataset in the AMBER format can be loaded directly into Chimera at a rate of about 2 configurations per second and into VMD at about 5 configurations per second. Both tools offer the benefit that no preprocessing needs to be performed, but this also means that the user must wait 15 (or seven in the case of VMD) minutes before starting to work with the dataset as a whole. The proposed streaming approach requires a preprocessing step, but accelerates the loading to more than 45 configurations per second. The performance of Chimera furthermore degrades so much while loading that it becomes unusable. This limitation does not apply to the new multi-threaded implementation as it is designed to continually stream data, thus not forcing the user to wait until the loading process is complete.

Table 3.3 shows the resulting performance values for the direct rendering of all molecules as individual glyphs. For other visualization modes, the values for fps and reloading vary strongly over time, as all the derived data changes constantly in density. However, since these modes generate less graphical primitives (molecules of clusters are omitted and only one ellipsoid is drawn per

cluster) the frame rates increase. The reloading values vary for a single dataset on different visualization modes because the amount of calculations performed at loading time differs. However, these variations are negligible, except for the pathlines, where the reloading rates can decrease down to 90% in most cases and 25% in the worst case (R-152a) due to the unoptimized implementation. This was not further investigated as it was only a stepping stone for the development of the flow group visualization. Because of the data streaming mechanism, the sizes of the data files on disk are irrelevant and the memory footprint is very small. Table 3.3 only presents the average size in memory of one configuration for each dataset. The complete file size of the datasets is shown in table 3.4.

The performance tests were executed 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 512^2 for all tests. All files were stored on the local SATA hard drive (no RAID hardware was used).

Table 3.4.: Preprocessing times for all nucleation datasets. The total time is given in hours and minutes, while the time per configuration is given in minutes and seconds.

dataset	file size	total time	time per configuration
10,000 methane	2.6 GB	0:37:00	0:00.4
50,000 methane	24.9 GB	21:23:00	0:07.7
R-152a	8.7 GB	29:42:00	1:46.9

Table 3.4 shows the preprocessing times needed to prepare the datasets. The table does not show the times for the TEM-1 dataset, because this being no nucleation simulation, no derived data must be calculated and the preprocessor is only needed to convert the file format, which is almost as fast as simply copying the file over network. When preprocessing the three nucleation data files, however, the derived data has to be calculated in a time-consuming process. Since the datasets have trajectories of different length, Table 3.4 also shows the average calculation time per configuration, for better comparability. While the times of the both methane nucleation datasets are quite acceptable, the R-152a dataset needs a conspicuously long time to be preprocessed. This is due to the special structure of the data, where after a short period of time almost all molecules are clustered in many rather small clusters, making the cluster tracking, for example, very expensive. These preprocessor runs were performed on a single machine with an AMD Opteron 248 processor running at 2.2Ghz with 4GB RAM. Compared to the simulations calculated on cluster computers with multiple processors and still needing several days, up to multiple weeks, the preprocessing time can still be considered acceptable. However for production

use the whole approach will have to be rewritten to be more efficient – which is part of the roadmap for MegaMol, the framework that succeeds the current point cloud tool and is being implemented in the context of the SFB716 by Sebastian Grottel. A brief example of the hypothetical workflow with the current tool is outlined below.

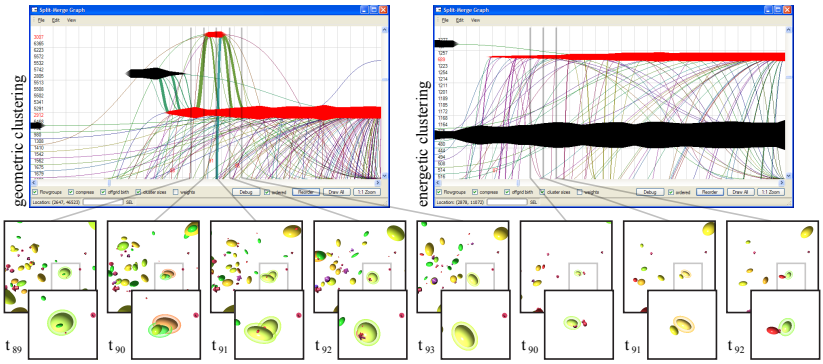


Figure 3.23.: Schematic views of two clustering algorithms applied to the 50,000 methane nucleation dataset. The top left image shows the results of a pure geometric clustering, and the top right image shows the results of a clustering based on energy levels. The lower images show zoomed-in views of the selected cluster at different configurations. The geometrical clustering not only detects too large and too many clusters, it also splits one cluster into two for three configurations. The flow groups' colors are based on the cluster IDs e and v, such that flow groups between pairs of clusters are colored identically for easier visual tracking.

Playing back the trajectory as animation reveals flow groups temporarily taking the place of small clusters, as shown in figure 3.19, which indicates a problem with the cluster detection. On the other hand, bigger clusters are always surrounded by some flow groups because many molecules first hit a cluster, but cannot join it immediately because of too different speeds and energy levels. They rebound, get slowed down and then join the cluster some configurations later. However, it is currently not clear if this should be considered an error in the classification or if this is a valid effect needed to obtain meaningful results. Such effects are much easier to discover when making use of a visualization tool, but to further clarify the observed situation, the molecule rendering alone is insufficient. The schematic view of the clusters' evolutions provides additional insight.

Figure 3.23 shows two schematic views of the 50,000 methane nucleation data-

set comparing a cluster detected with two different algorithms. The algorithm employed in the left view uses only the simple geometrical distances between molecules to define adjacencies. A molecule with a pre-defined number (normally four) of such neighbors seeds a cluster. The algorithm applied in the right view defines two molecules as clustered if the sum of their potential energy and their relative kinetic energy is negative [Hil55]. This energy-based approach yields far better results when extrapolated to experimental values [Kie06]. The geometrical clustering not only detects too many and too large clusters, but also often creates multiple close-by clusters instead of a single one. The small red cluster S_{3007} at the top of the left schematic view is such an example. The ends of this cluster are connected to the lower and bigger red cluster S_{2912} with thick flow groups indicating that almost all molecules of S_{3007} came from and rejoin S_{2912}. The clustering with the energy-based algorithm shown in the right image does not exhibit such splintering clusters. Here, the cluster corresponding to S_{2912} is cluster S_{689}, which is rather constant in its size compared to the clear dent in S_{2912} in configuration t_{91}. There are only very small flow groups of only one or two molecules, which is characteristic for normal fluctuation between adjacent clusters.

Another interesting situation is given on the left side of the left schematic view in figure 3.23, where the black cluster S_{2805} feeds cluster S_{2912} before it vanishes. Without the molecule view, the reason for this issue can hardly be identified. With the coupled view, one can see that cluster S_{2805} got rather slim and long around configuration t_{88} so that the geometrical approach failed to find the required four neighbors for one molecule to detect the cluster. Similar situations can be observed in the smaller 10,000 methane nucleation dataset. A small part of the graph of the geometric cluster detection for this dataset is shown in figure 3.21.

All these findings are dataset-dependent, as figure 3.24 shows. The nucleation of the refrigerant Difluoroethane (R-152a) results in most molecules clustered in many rather small droplets. Almost no fluctuation between the clusters takes place, however, except for rebounding molecules forming small flow groups as described above and for the merging of clusters, clearly indicated by large flow groups in the schematic view. Any errors in the cluster detection could therefore be spotted rather quickly. However, for this dataset the energetic clustering produces very good results.

The information provided by the coupled visualization has been put to good use at the ITT. The inadequacies in the classically employed geometric clustering algorithm used previously could be pinpointed. Subsequently, the improved energy-based algorithm based on the knowledge of the failure situations of the existing clustering algorithm has been developed. However, in some situations also the pure energetic clustering delivers incorrect results. Thus, a combination of these approaches was created, showing promising results in several simulations. It is still being investigated, however, if the hybrid

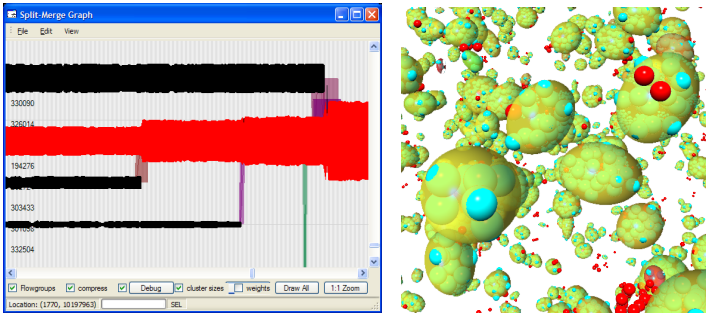


Figure 3.24.: Schematic view (left) and molecule view (right) with transparent cluster ellipsoids and monomers (red; clustered molecules cyan) of the R-152a nucleation simulation. Almost all molecules are very quickly clustered in many rather small and stable clusters (appearing yellow in this color-scheme). Greater changes only happen when two clusters merge, which is clearly indicated by the corresponding thick flow groups in the schematic view.

approach can reliably improve the results for arbitrary substances and parameters.

3.3. Tensor Data

During a collaboration with pracitioners in neurosurgery[3], we worked with diffusion tensor data sets, which can be interpreted as points with high semantic density. Tensors in general can as well be seen as a generalization of scalars, vectors and matrices: In k dimensions, a tensor has n indices and k^n components; n is also called the *rank* of a tensor. Thus, scalars are tensors of rank $n = 0$, vectors of rank $n = 1$ and matrices of rank $n = 2$. Diffusion tensors represent the diffusion characteristics of hydrogen in organic tissue. This is of special interest since the underlying cell structure influences these properties such that strongly aligned cells restrict the diffusion to an anisotropic behavior. This information is very useful for medical scientists when the human brain is concerned. As important neuronal pathways feature these anisotropic cell characteristics, one can infer about the occurrence of such major *white matter* tracts based on diffusion tensor data [Bea02]. The protection of these white matter tracts is of utmost importance during intervention lest the patient suffer irreparable damage and consequential physical and or psychical dysfunction. Therefore, visualization of the tracts is getting increasingly relevant to diagnosis and intervention planning. However, many visualization methods are not capable of representing all tensor information in a comprehensive manner.

Diffusion tensor datasets are acquired by magnetic resonance imaging (MRI), which usually generates information about the quantity and linkage of hydrogen in the matter that is examined. Diffusion tensor imaging (DTI), as an MRI technique, on the other hand, can generate diffusion information as well. For the computation of diffusion tensor data, six diffusion-weighted images with different gradient directions are measured in combination with a reference image, measured without any gradient. Based on this set of images, the real-valued symmetric second-order tensor can be determined, averaged over the corresponding volume, for each voxel [WMM+02]. The result is a 3D regular grid with tensors at each point. The diffusion tensor is a symmetric 3×3 matrix, hence its eigenvalues are real and the corresponding eigenvectors are perpendicular. The major eigenvector represents the direction of the diffusion. Direct visualization of its components on an arbitrary volume slice is a well-known method for diagnosis purposes [MRIB00, PAB02], but very difficult to interpret. Another approach highlights the anisotropy of the eigenvalues, that is how much of a directional flow occurs in any one grid point, using a grey scale map [UBBvZ96]. It has also been proposed to use a color map of sphere normals to better emphasize the direction of the flow. Depending on the direction of the view, one of the r, g, b colors indicates clearly those tensors that point away from the slice plane [PP99]. For a more comprehensive visualization of DTI data, volume rendering was proposed by Kindlmann *et al.* [KW99, KWH00] who utilized textures and transfer functions to repre-

[3]Neurocenter, Department of Neurosurgery, University of Erlangen-Nuremberg

sent the tensor properties. To investigate the features of the tensor per voxel directly, the eigenvalue/eigenvector pairs can be used to scale and orient a primitive or only certain features of a primitive, since it is canonical to interpret the eigensystem as the local glyph coordinate system. Using the eigenvectors as main axes for an ellipsoid and the eigenvalues as radii, all features of a tensor can be represented [vWPSP96]. Using a composite glyph of one ellipsoid per eigenvector instead of a single one in total has also been proposed [WMM+02]. A Haber glyph – a composite of cylinders – also exhibits enough features to map all of the eigenvalues to it [Hab90]. To reduce the ellipsoid ambiguity depending on the view point/orientation, Kindlmann proposes a different parametrization and uses superquadrics as a basis [Kin04a]. Superquadrics only have one radius, but their shape can be parameterized to convey the other values of the eigensystem by varying the geometry between spheres, cylinders, and cuboids. This results in features that can be more easily perceived regardless of the orientation.

All of these glyphs represent each tensor independently by shape, size, and color and are placed at the corresponding voxel position. These direct visualization techniques are useful for a detailed inspection of the DTI dataset. However, for medical staff their weakness lies in the missing connectivity of the data which prevents the analysis of complete pathways, or white matter tracts. The first step to the visualization of white matter tracts is the analysis of the MRI data to identify them. Streamline tracking techniques were adapted to DTI processing by different researchers [BPP+00, LAS+02] to provide a solution. A combination with advection to stabilize the propagation has also been proposed[Kin04b]. The drawback of streamlines in the context of DTI is certainly the restriction to a vector field. Features of the tensor field like torsion or the medium and minor eigenvalues cannot be displayed with this method. To improve the visualization of fibers, the streamlines were extended to stream tubes [ZDL03], which are rendered using elliptic profiles and can thus also display the two medium and minor eigenvector/eigenvalue pairs. However, the medium eigenvector is scaled so that effectively only the ratio between the medium and minor eigenvector remains intact, allowing the user only to distinguish tubes by their relative “flatness” or “roundness”. These stream tubes are also called Hyperstreamlines [DH93, WL01]. Neuronal pathways can also be extracted and visualized as surfaces [ESM+05], however the main focus in this context is on the extruded ellipse-profile lines. In the following an approach for the GPU-based rendering of such primitives is described.

3.3.1. Data Processing

The first step towards generating streamlines is the tracking of the related streamlines. Integration schemes known from flow visualization, such as Euler or Runge-Kutta, are applied to determine streamlines through a vector

field. The required input vector field is derived by a simple reduction of the tensor to its principal eigenvector. The loss of information is partly compensated by introducing a threshold for tracking. One possible parameter is *fractional anisotropy* (FA), which formalizes how much the eigenvalues are anisotropic, normalized to the unit cube [BP96]. It serves as stop criterion to terminate tracking when running into areas of reduced anisotropy with low probability for neuronal pathways. To avoid strong fluctuation of the tracking in areas with reduced anisotropy, the tensorline approach could be used instead [Kin04b], for example, but this would of course change the semantics of the displayed streamlines.

For the shown datasets all voxels above the specified FA threshold (0.3) were used as seed regions. As soon as the tracking enters a voxel with an FA value below the threshold it stops. A resulting whole-brain tracking can be seen in figure 3.25. Afterwards, subsets of fibers were selected using manually defined regions of interest. Each fiber is a collection of voxels with the corresponding tensor data. To be able to rely on an established implementation for the data management and fiber tracking, the presented approach was implemented using the MedAlyVis[4] framework, effectively adding a fiber-specific rendering back-end.

The presented approach is a novel technique that makes use of programmable GPUs to generate stream tubes/hyperstreamlines from a terse set of parameters directly on the GPU. The eigenvalues and eigenvectors of the tensor at each base point of the streamline serve as basis for the twist and the semimajor axes of the segments of the hyperstreamline. An oriented ellipse is generated at the sampled points, and all ellipses are then linearly interpolated to form tubes connecting the voxels. Together the tube sections form hyperstreamlines as shown in [DH93]. Instead of creating a mesh from these tubes on the CPU, the orientation and size parameters of each "tubelet" are uploaded to the graphics card. Rendering takes place using sphere tracing [Har96] to ray cast the surface on the GPU. This is motivated by the fact that hyperstreamlines in medical visualization are most useful when displayed in the context of a volume representation of the surrounding tissue. If the data transfer from CPU to GPU is minimized, the system bus bandwidth can be used to upload volume data in case of time-dependent or bricked datasets. Another advantage of this approach is that it does not access textures at all, so the volume visualization can use all of the available bandwidth on the graphics card to keep performance as high as possible. On the other hand, the rendering of hyperstreamlines relies heavily on the computational power that is available on current GPUs but is not needed for direct volume rendering approaches.

An approach very closely related to the following was developed by Stoll *et al.* [SGS05], however the tubes generated by their method are limited to cir-

[4]Developed at the Neurocenter Erlangen in the context of project C9 in the SFB 603

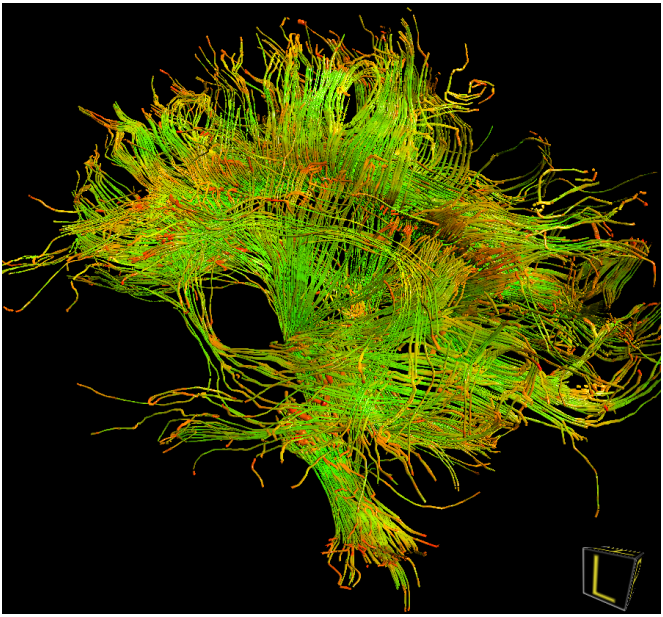


Figure 3.25.: Hyperstreamline visualization of a whole-brain tracking. The viewpoint is positioned left, posterior above the brain. Green segments indicate tracking in reliable regions with high anisotropy. Yellow tube sections suggest fiber crossings. The absence of red segments in the central parts of the hyperstreamlines documents the correctness of the fiber tracking algorithm. This dataset is as used for performance measurements in Table 4.1.

cular cross-sections as opposed to the proposed elliptical cross-sections. This trade-off allows for the rendering of the resulting tubes using splatting techniques that yield much higher frame rates than our approach at the cost of a lower semantic density.

3.3.2. Tubelets

To visualize the tensor field, the data is mapped to a tubelet shaped by ellipses within a slice plane at each end. These ellipses are defined by their semimajor and semiminor axis and its rotation around the local x axis. The shape along the tubelet is determined by linearly interpolating the ellipse orientation, as further described in Section 3.3.4. Taking into account the special needs of

our given data, a local coordinate system is introduced in Section 3.3.3, and some more complicated customizations on distance measuring and rotation interpolation are required as described in Section 3.3.5.

3.3.3. Definition of the Local Coordinate System

The local coordinate system for each tubelet is defined depending on the two ellipses that control the tubelet's shape. The ellipses are defined by an eigensystem each, where the normalized eigenvectors $\vec{e}_i$ are sorted by their corresponding eigenvalues v_i in descending order: $v_1 > v_2 > v_3$. As $\vec{e}_1$ is the direction in which the hyperstreamline has been tracked, the medium and the minor eigenvalue define the length of the ellipse's semi axes and the corresponding eigenvectors define the corresponding direction of each semi axis. The signs of the eigenvectors are chosen in a way such that the eigensystem is a right-handed orthonormal basis that can be used directly to define the tubelet's local coordinate system.

The local x-axis is given by the tubelet's axis $\vec{x}_t$, connecting the centers $\vec{p}_l$ and $\vec{p}_r$ of the two ellipses ($\vec{x}_t = (\vec{p}_r - \vec{p}_l) / \|\vec{p}_r - \vec{p}_l\|$), and the other two axes are derived from the left end ellipse as follows: The ellipse's first eigenvector $\vec{e}_1$ is to point in the same direction as the tubelet axis. This is done by rotating the eigensystem around $\vec{e}_1 \times \vec{x}_t$ by an angle of $\arccos(\vec{e}_1 \cdot \vec{x}_t)$. Then the rotated eigenvectors $\vec{e}_2$ and $\vec{e}_3$ define the tubelet's local y-axis $\vec{y}_t$ and z-axis $\vec{z}_t$ respectively.

3.3.4. Geometrical Definition of the Tubelets' Shape

Geometrically defining the shape of a tubelet first requires some considerations about ellipses in the yz-plane: Assuming r_1 and r_2 to be the two semiaxes, ellipses are usually described by the Cartesian equation

$$\frac{y^2}{{r_1}^2} + \frac{z^2}{{r_2}^2} = 1 \,. \tag{3.17}$$

Our ellipses are defined within the yz plane and may be rotated around the x axis. Therefore a representation of the ellipse corresponding to the polar coordinates of a circle is easier to handle (see figure 3.26). Additionally, the surface normal is needed later for correct lighting, which is also easier to calculate in polar coordinates. Let φ be the angle to the y axis, then the ellipse can be given in polar coordinates by

$$\vec{x}(x, \varphi) = \begin{pmatrix} x \\ r(\varphi)\cos\varphi \\ r(\varphi)\sin\varphi \end{pmatrix} \tag{3.18}$$

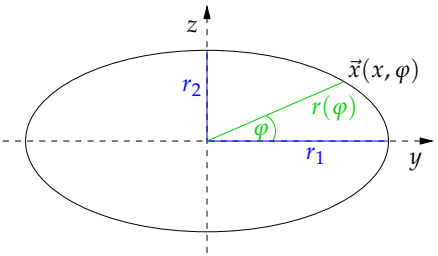


Figure 3.26.: Ellipse in polar coordinates.

with $r(\varphi)$ determined by plugging these coordinates into the Cartesian equation (3.17)

$$r(\varphi) = \frac{r_1\, r_2}{\sqrt{r_1\, \sin^2\varphi + r_2\, \cos^2\varphi}}\,. \tag{3.19}$$

Within a plane of constant x, rotating the whole ellipse around the x axis by an angle ρ changes (3.18) to

$$\vec{x}(x,\varphi) = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ r(\varphi)\, \cos(\rho+\varphi) \\ r(\varphi)\, \sin(\rho+\varphi) \end{pmatrix}\,. \tag{3.20}$$

The tubelets are defined by an ellipse at each end and are centered along the x axis of their own local coordinate system. These ellipses may vary in the length of their semi axes and in the rotation along x, and they are located at $(-l/2, 0, 0)$ and $(l/2, 0, 0)$ respectively, where l is the length of the tubelet as depicted in figure 3.27. For each x value along the tubelet the semiaxes and the

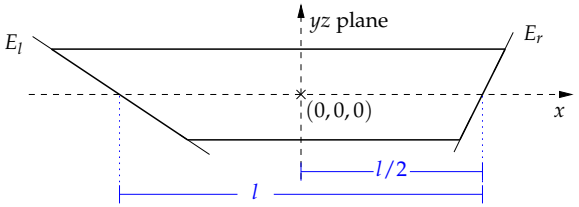


Figure 3.27.: Definition of a tubelet along x.

rotation angle ρ of the ellipse are calculated by linear interpolation from the properties of the ones at the tubelet's ends.

The tubelets are connected to each other to get longer tubes, so the cutting planes E_l and E_r at the ends of each tubelet have to be specified. In order to approximate curved tubes using linear tubelets, these planes may be rotated arbitrarily as illustrated in figure 3.27.

3.3.5. Geometrical Background for Ray Casting

In order to ray cast each tubelet's surface the intersection point of the eye ray with the tubelet is needed. As this intersection calculation leads to an equation of degree 4, it is very hard to calculate analytically and would require numerical solving methods, too expensive and time-consuming for graphics hardware. Therefore the sphere tracing algorithm presented in [Har96] was adopted. This approach has been recently reproposed for displacement mapping on graphics hardware [Don05], where the analytical calculation of the ray-surface intersection would be very hard, as happens for the extruded ellipses that are used here. Starting at the eye's position an iteration along the eye ray takes place, determining the current distance to the tubelet's surface after each step, and using this distance as the next step size to close in onto the tubelet. This procedure is repeated until the distance falls below a specified threshold ω. In most cases this works fine, given sufficient iterations, but some rays require additional adjustments which will be explained in Section 3.3.6. From the current position, the shortest distance to the tubelet has to

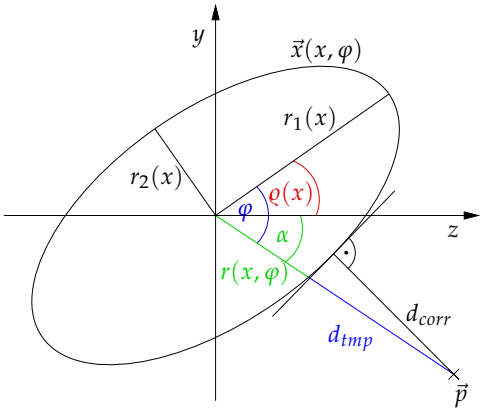


Figure 3.28.: Distance measurement from the current position $\vec{p}$ to the rotated ellipse $\vec{x}(x,\varphi)$ within a plane of constant x.

be calculated to get the new step size. Due to the rotation of the tubelet along x, the correct shortest distance d_{corr} to the tubelet's surface – which is measured along the surface normal $\vec{N}$ – cannot be easily calculated on the GPU as

it would also lead to an iterative and time consuming way to find the solution. To avoid that, the shortest distance is replaced by an approximated solution: assuming that the x value is the same as the one of the current position, the distance d_{tmp} to the ellipse in this common plane normal to the x axis and through the current position can be calculated, as depicted in figure 3.28:

$$d_{tmp} = \sqrt{\vec{p_y}^2 + \vec{p_z}^2} - r(x, \varphi) \tag{3.21}$$

where $\vec{p_y}$ is the y-component of $\vec{p}$ and $\vec{p_z}$ the z-component respectively, and with r satisfying (3.19) within the current plane of constant x value.

In the next step the current position is moved d_{tmp} units further along the ray. In case of $d_{tmp} < 0$ the surface has already been intersected. However, stepping backwards is equivalent to walking along the ray in negative direction and needs no special considerations.

In order to get a longer tube which can show more data values than a single tubelet, neighboring tubelets are connected at their cutting planes E_l and E_r. To approximate a curved tube the planes do not necessarily have to be normal to the local x axis, as mentioned before. These sloped tubelet ends require further considerations about the tubelet's properties though: merging the tubelets to form larger tubes raises the problem of rotation consistency at the interconnections. Consistent properties of the ellipses are only guaranteed in slices perpendicular to x_t that do not contain parts of a neighboring tubelet, because otherwise there are two inconsistent interpolation parameters belonging to the two tubelets respectively. To solve this, the interpolation is restricted up to the point $\vec{q}$ as exemplified in figure 3.29 for the tubelet's left end.

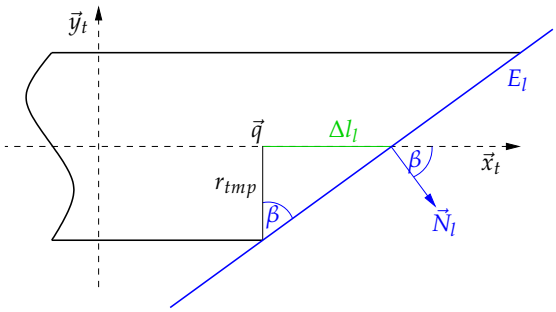


Figure 3.29.: Correction of distance and interpolation range. E_l is the left cutting plane, $\vec{N}_l$ the corresponding normal vector.

To restrict the interpolation to the correct range of x values, the distance $\Delta l = r_{tmp} \cdot \tan\beta$ must be calculated, with β being the angle between the tubelet's local z-axis z_t and the intersection line of the cutting plane and the plane normal

to the local x-axis x_t, which is $\beta = 0$ for the left end and $\beta = \rho_r$ for the right end – due to the definition of the local coordinate system as described in 3.3.3 – and r_{tmp} satisfying (3.19) with $\varphi = \beta - \rho(x)$ leading to

$$r_{tmp} = \begin{cases} r_{1l} r_{2l} / \sqrt{r_{2l}} & \text{(left end)} \\ r_{1r} r_{2r} / \sqrt{r_{1r} \sin^2 \rho_r + r_{2r} (\cos^2 \rho_r)} & \text{(right end).} \end{cases}$$

Taking this fact into account for the computation of the tubelet's total length the result is

$$l_{total} = l + \Delta l_l + \Delta l_r \tag{3.22}$$

with $\Delta l_l , \Delta l_r \geq 0$.

To enhance the impression of the tubelet's surface shape the correct surface normals are needed for correct lighting. Using the derivatives of (3.20) we get the surface normal $\vec{N}'$ by evaluating

$$\vec{N}' = \frac{\partial \vec{x}}{\partial \varphi} \times \frac{\partial \vec{x}}{\partial x} . \tag{3.23}$$

Shading will be further detailed in the following section.

3.3.6. Sphere Tracing on the GPU

To generate tubelets on the GPU, the desired surface is reduced to its parameters as described in Section 3.3.2 and uploaded to the GPU. The vertex program calculates a bounding cuboid to scale the rendering primitive (a point) accordingly. Most of the values that are constant for a whole tubelet are computed from the parameters as well, however the amount of attributes that can be passed to the fragment program are limited, so the implementation for the DirectX 9 generation of hardware suffers from a couple of calculations in the fragment program that would have been optimal in the vertex program. The additional data is passed to a fragment program along with the original parameters so it can find the proper surface intersection with one ray cast per fragment, add Phong shading and correct the depth value to fit the geometry.

For the GPU-based part of the algorithm the position, two colors, orientation (as a quaternion), the four radii, the right end rotation angle, the total length and the normals of the two bounding planes are needed. They can be fit into five float quadruples (O_{local} and $l, q, r_{1l}, r_{2l}, r_{1r}, r_{2r}, \vec{N}_l, \vec{N}_r$ and ρ_r, as ρ_l is always 0 due to the definition of the local coordinate system) plus two byte quadruples (color), so only 644 bytes are uploaded per tubelet, which is less than what is needed for 10 triangles with normals and constant color. For each tubelet a single point (with the attributes as texture coordinates) is sent to the graphics card, since a point is the smallest type of billboard in terms of data size, and as

bonus it is not even necessary to adjust the orientation to face the eye position $\vec{p}_e$.

The vertex program computes two orientation matrices. The first one (M_c) is obtained after combining the tubelet orientation quaternion with the camera orientation quaternion. It is used for orbiting the eye point around the tubelet, to obtain the local coordinate system described in Section 3.3.3. The second matrix (M_o) is obtained from only the tubelet orientation and used to calculate a bounding cuboid from the worst-case dimensions of the tube, i.e. a cylinder with length l_{total} and radius $max(r_{1l}, r_{2l}, r_{1r}, r_{2r})$. This cuboid was chosen to make sure that no part of the tubelet lies outside the billboard with the perspective projection that is used. This cuboid is projected onto the view plane to obtain the point's extents and center. Since the light position is also constant for all pixels of one tubelet, the light vector of the single light source is rotated with M_c in order to always have a headlight-like illumination.

The parameters passed to the fragment program are the following: the eye position $\vec{p}_e$ relative to the tubelet centered at $(0, 0, 0)$, the rotation matrix from the combined quaternions M_c, the transformed light vector, the two bounding planes and the radii and length. Furthermore $\Delta l_l, \Delta l_r$ need only be calculated once for each tubelet so they are computed in the vertex program and passed to the fragment shader. Together with the re-oriented z vector $\vec{o}'$ all available varying parameters shared between vertex and fragment program are exhausted.

Analogously to the previously presented primitives, the fragment program first has to find the vector $\vec{s}$ which connects the eye to the current pixel starting from the x and y component of the fragment's *window position* $WPOS$. To account for the fact that the origin is at the tubelet and the eye point orbits around it, M_c has to be applied to the resulting normalized ray direction $\vec{s}$ as well.

To speed up the iteration process, an approximation of the intersection point is used as a starting point: The tubelet's surface is enclosed between two conical frustums, interpolating between the major axes of both ellipses in the bigger frustum and interpolating between the two minor axes in the other one. The intersection of the eye ray with the two cones is calculated and the middle point between both intersection points used as the starting point for ray casting.

Then the iteration along the ray with a step size computed from the approximated distance as in (3.21) takes place until either the distance is below a threshold ω or a maximum number of steps has been walked (see below). If the distance left after the last step is beyond this threshold or the current position is outside the two clipping planes E_l, E_r, the fragment is discarded. This leads to holes in the surface if the ray is approximately parallel to the tubelet's axis as the step size is almost constant and often very small, but it might still be far from the intersection point with the surface. To enhance the results in that

case, an adaptive step size is used: if the angle between $\vec{x}_t$ and $\vec{s}$ is small, the step size is scaled according to the angle between the surface normal and the ray direction by $(1 - \cos(\vec{N} \cdot \vec{s})) + 1$ which avoids these holes and thus leads to much better results with less iteration steps.

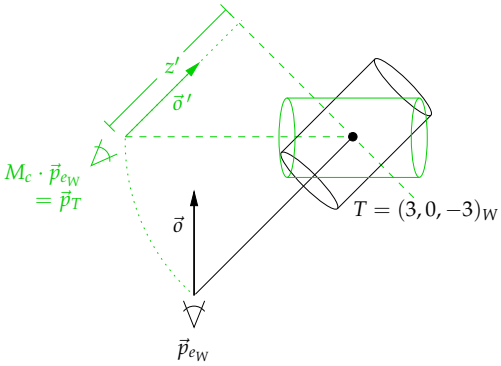


Figure 3.30.: Local tubelet coordinates (green) in relation to world coordinates (black).

In case the fragment is not discarded, the correct depth is calculated to ensure that tubelets intersect correctly with each other and the volume as well. Since the eye point is displaced from $(0, 0, 0)^T$ and the view direction is no longer $\vec{o} = (0, 0, -1)^T$, the depth z' is the distance to a plane normal to the orientation transformed by $\vec{o}\,' = M_c^T \vec{o}$ (see figure 3.30), the Hessian normal form can be used to get the distance and then fit the result to the exponential OpenGL depth range:

$$\begin{aligned} z' &= \vec{o}\,' \cdot (\vec{p} - \vec{p_e}) \\ z_{ogl} &= -\frac{z_F + z_N}{2(z_N - z_F)} + \frac{1}{2} + \frac{z_F z_N}{(z_N - z_F)z} \end{aligned} \tag{3.24}$$

The normal is calculated as defined in (3.23) and used for Phong shading of the surface.

A drawback of this technique is that one cannot look inside the tubelets, since for rendering the back face a starting point on the other side of the local x axis would have to be used and thus the rendering would require two times the performance needed by the current implementation. However there is no need to render the back face since users are usually not looking down the length of a tubelet because the relevant information (how the rotation and radii change over a certain distance) is visualized along the length of the tubelet and has to be interpreted in the context of the local x coordinate.

3.3.7. Results

The advantage of DTI is its capability to provide the intrinsic diffusion properties of water within tissue. Due to the anatomical structure of neuronal pathways this diffusion is anisotropic in areas of major white matter tracts. Thus, DTI can reveal coherences in-vivo which are not visible in MRI_{T1} or MRI_{T2} datasets[BML94]. An accepted method to access this information is to apply fiber tracking. However, since streamlines cannot convey tensor information their extension to hyperstreamlines is of certain value for detailed data analysis.

Figure 3.31 (left) shows a line-rendering in comparison to hyperstreamlines (right) of a pyramidal tract combined with direct volume rendering of a MRI_{T1} dataset. Using hyperstreamlines instead of simple lines enables the analysis of the whole tensor data. Areas with large eigenvalues will result in larger diameters of the hyperstreamline. This allows users to make conclusions about the underlying tissue. For a hyperstreamline showing a higher diameter it is very likely that it is not aligned with a neuronal pathway since white matter restricts the diffusion perpendicular to the cell orientation. Therefore, such expansions are an indication for an uncertainty in the fiber tracking.

The analysis regarding uncertainties can be further improved by the application of special color schemes. Instead of utilizing the standard RGB-scheme, where the principal eigenvector is used as color vector, the color can be selected by a scheme based on an approach presented by Westin *et al.* [WMK+99]. Thereby, areas with high anisotropy are depicted green while areas with planar diffusion are colored yellow and isotropic areas appear red. Accordingly, red tube segments do have a higher uncertainty. Figure 3.25 shows a whole brain tracking. It can be seen that especially in the end segments the color changes from green to red. This is plausible since the fiber tracking algorithm stops when reaching a voxel with a sufficiently low anisotropy which is depicted red.

Segments which appear yellow correspond to regions of planar diffusion. This occurrence is generally considered to be a potential fiber crossing which cannot be treated adequately with current fiber tracking algorithms. Therefore, such segments are of special interest and supportive rendering is desired. For the actual analysis of such regions hyperstreamlines are superior to common streamlines and -tubes. They provide information about the spatial orientation of the diffusion (figure 3.32 *left*). However, hyperstreamlines suffer from the same problem of restricted data accuracy as standard streamlines. Since DTI data does not provide better resolution than the current 2mm voxel spacing, all tracked lines can only be considered averaged models for the underlying tissue structure.

Another feature which is depicted by hyperstreamlines is torsion (Figure 3.32 *right*). While strong torsion is not necessarily required for DTI visualization it

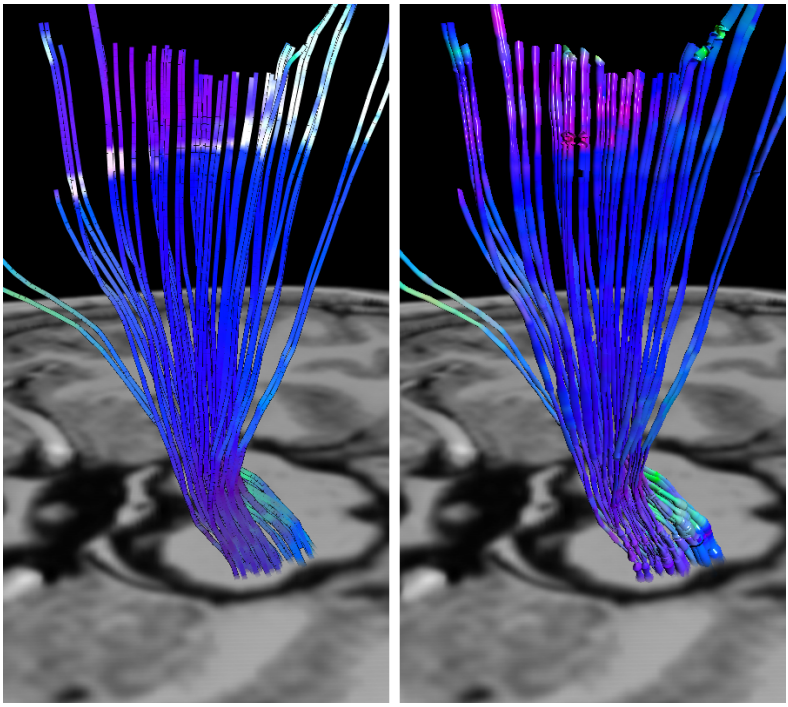


Figure 3.31.: These two figures show the same pyramidal tract rendered with standard streamline (left) and with the proposed method (right) in combination with direct volume rendering of a T1-weighted MRI dataset. For coloring the principal eigenvectors are mapped into RGB-color space.

is extremely useful for other data, namely technical simulation data.

3.3.8. Performance

The GPU-based tubelets have been integrated into a framework for DTI visualization used at the Neurocenter for easy access to MRI data preprocessing and rendering of correct context information as well as the possibility to compare the novel approach with existing streamline/streamtube implementations. Two variants of GLSL high-level shaders have been implemented: the first avoids Shader Model 3 functionality and walks a fixed number of steps before testing for a hit, while the second uses a dynamic loop that stops if either ω can be satisfied or a certain number of steps is exceeded. For DirectX

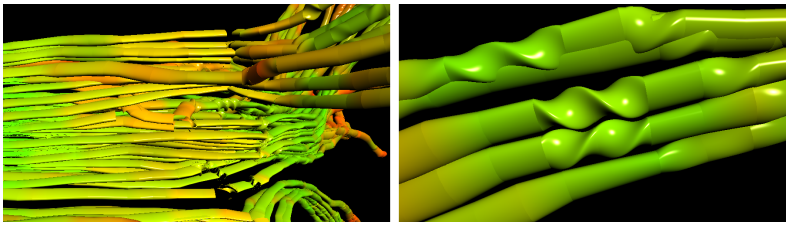


Figure 3.32.: The left figure shows a bundle of hyperstreamlines traversing a region of planar diffusion which leads to yellow coloring and a flattening of the tube. On the right side, some segments showing extreme torsion are depicted. The displayed extreme torsions are added manually as a proof of concept by switching off angle correction, thus allowing rotations larger than 90 degrees between consecutive ellipses.

9 generation hardware the second approach cannot improve performance significantly, since fragments in a certain neighborhood are required to have the same execution time, so a 'slow' fragment defeats any time-saving neighbors. This variant can however be used for investigating the relation between surface curvature and number of steps needed to intersect the surface reliably (see figure 3.33). For practical purposes, a fixed iteration bound of 8 works sufficiently well and yields a performance of about 33 MPixels/s according to *NVShaderPerf* [NVS].

Figure 3.33.: Iteration count to satisfy threshold shown as hue where red means higher iteration count.

To sensibly compare the resulting performance to an existing polygon-based approach, some peculiarities have to be considered. The polygon tubes are subdivided into 16 segments per profile in order for the surface to have comparable shading quality. The same number of tube segments is used in both approaches, however the existing algorithms linearly interpolate between the profiles, which yields connections that are not really correct (see figure 3.34). To obtain a high-quality surface like with GPU-based raycasting, the segments would have to be subdivided in relation to the spanned rotation angle, which would yield at least three times the number of primitives that are used cur-

rently and thus reduce rendering performance drastically.

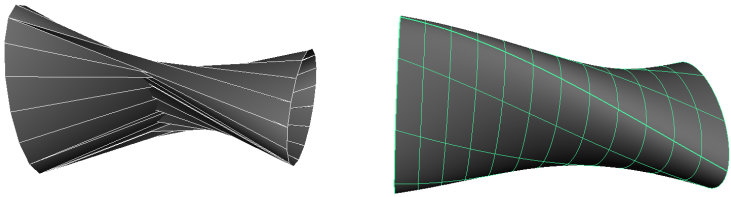


Figure 3.34.: Interpolated vertex positions (left) with erroneous self-intersection vs. interpolated ellipse orientation (right).

As can be seen in Table 3.5, tubelets cannot offer the performance of the much simpler polygon-based approach for small and medium-sized datasets. With the whole-brain tracking, however, we are coming close to the break-even point, because the high load of the geometric primitives on the CPU and the graphics bus has about nullified the advantage of cheap fragment processing, so the novel approach is about twice as fast with a medium-sized viewport on a GeForce 7800 GTX. However, it is also limited by fragment processing power, which means that using a nearly full-screen-sized viewport reduces our frame rate to half the frame rate achieved by the polygon-based approach. This can be solved by several means though, since fragment processing power can be much more easily increased than throughput or geometry processing: either two SLI-coupled graphics cards could be used for about twice the performance or one could resort to distributed parallel rendering as often employed in direct volume rendering. Taking into account the recent development of graphics cards, one can also rely on the fragment units of the next generation of graphics cards offering at least about double performance through higher parallelism and other optimizations.

It is also evident in Table 3.5 that standard direct volume rendering does not significantly impact the performance of the proposed approach. Nevertheless, using the instrumented driver the suspicion arose that there still must be a bottleneck in the implementation since the fragment shader load is maxed out at 74% while other raycasting algorithms, like the dipoles in section 3.2.2, obtain up to 91%.

	Optic Tract	Pyramidal	Brain
#segments	18,681	34,778	226,032
fps 434×673	10	10	6
fps 434×673 with volume	10	10	5.5
fps 1560×1051	2.5	2.3	1.5
fps 1560×1051 with volume	2.4	2.3	1.5
fps polygons	> 100	> 100	4
fps polygons with volume	15	15	3.3

Table 3.5.: Performance measured on a GeForce 7800 GTX with instrumented developer driver v79.70. Viewport sizes are as indicated, with the tubes zoomed to fit. A contextual volume rendering is included where mentioned. For polygon-based visualization the viewport size is always 1560×1051, with the volume approximately filling it. Each polygonal segment consists of a triangle strip of 21 elements.

3.4. Conclusion

This chapter demonstrated that carefully optimized GPU-based raycasting of various primitives is a valid alternative to polygon-based rendering. It is a hybrid image/object space method that benefits from the quality of raycasting, while containing the cost through the usage of proxy geometry that is used to trigger the raycasting. The main strength of this approach, besides the high rendering quality, is the low upload bandwidth required, which makes it especially useful for extremely large and/or time-dependent datasets. The main drawback is of course the high shader load caused by the more complex glyphs, which limits the interactively usable viewport size somewhat, depending on the GPU performance. This can however be counteracted by the use of several GPUs to subdivide the workload in image space.

Even with very complex glyphs there are still datasets that contain too many attributes to be directly visualized. Methods for the visualization and mapping of such data will be presented in the next chapter.

4 Multivariate Data and its Representation

Unlike the data visualized in chapter 3, which exhibits a limited number of attributes per point/record, there are datasets which have such a high number of attributes that it is impossible or at least very difficult to visualize all of them directly without incurring into drawbacks that are as grave as the original problem, if not worse, especially if there are also thousands of points that need to be represented. A term has been brought forth to point out the difficulties with high-dimensional data: the 'curse of dimensionality' [Bel61]. To help putting this problem into perspective, the following example just takes into consideration the resolution limitation of current output devices. This oversimplifies a bit, but still points out the difficulties. If we just allocate one pixel per attribute per point [Kei00], on a full HD Display there can still be no more than 2 million items (1D, or a single attribute) or 40,000 items (50D). This of course assumes that all of the points and their attributes can be sensibly mapped to a color and discrete position and still retain some comparability and readability without having to resort to more complex representations per point. If icon-based techniques are used, i.e., some kind of shape which is parameterized with the available attributes, then the maximum number of items than can be inspected simultaneously drops much more quickly, since such icons need at least one order of magnitude more screen space to be interpretable. A viable alternative is to not explicitly depict all of the attributes, but to focus on the overall structure of a dataset and how the items relate to each other, that is, laying out the items by their similarity and displaying only a subset of their attributes, for example. The potential user should then be able to recognize existing clusters from the distribution of the rendered data. This chapter will elaborate on dimensionality reduction algorithms that allow for the generation of such a low-dimensional mapping of the original data and detail their combination with a GPU-based visualization of the results as well as an approach to exploit the GPU for the acceleration of the mapping step itself. Results will be shown for datasets from machine learning and medicine.

The work in this chapter is based on [RE04] and [RE05b] and some paragraphs and figures are excerpted from the original publications.

4.1. Related Work

In the past, different strategies for dealing with exceptionally high-dimensional data have been developed. On the one hand, several concepts have been devised to visualize the many dimensions on displays that are mostly limited to two. Scatterplot matrices are a classical example. The concept is simple: combine each dimension/attribute $d_i \in D$ with any other, and render the resulting 2D scatterplot at position (d_x, d_y) of a $|D|^2$ scatterplot matrix [BC87], the diagonal of which is empty as combining dimensions with themselves represents no information increase. The different coordinate axes can also be stacked, for example, either just laying them out in parallel along one axis and connecting the respective values with a polyline [Ins85], which mostly has the advantage of making the detection of similarity over several dimensions simple. Alternatively, the value ranges can be stacked along both available axes (x and y) by first discretizing the ranges of all dimensions and then recursively embedding them into each other [LWW90], using single pixels for each record. A variant that recurses on a single dimension but combines this with space-filling-curve patterns for the arrangement of the elements was proposed as well [KAK95]. The axes have also been arranged radially, using the resulting pie slices in between them to string together the elements according to the value of the respective dimension [AKK96]. This last approach is actually a combination of the pixel-based pattern techniques and the star glyph, where all items are laid out individually in a matrix and the attribute values depicted as the length of radially arranged 'spikes' [SFGF72]. There also is a middle-ground variant called 'star coordinates' with non-orthogonal radial axes for scatterplots [Kan01]. Other iconic representations include stick figures with irregularly arranged axes [PG88] and the Chernoff faces, where each dimension is mapped to a feature of a stylized cartoonish face (lips, eyes, eyebrows etc.) which is changed in size/deformation [Che73].

The orthogonal approach maps all of the data to a lower number of dimensions, usually two or three, which completely abstract from the actual attribute values and represent only the similarity between points by spatial proximity. This does obviously not preclude the use of glyphs for the individual items to display some of the more important attributes. Probably the most important dimensionality reduction strategy is multi-dimensional scaling (MDS), which exists in different variants. Classical MDS [BG97] minimizes a loss function called strain starting from a dissimilarity matrix, while metric MDS is a more general variant, which also supports weighted distances and a variety of loss functions. Non-metric MDS additionally supplies a monotonic relationship between the dissimilarities.

When employing metric MDS, the quality of the low-dimensional mapping can for example be determined by calculating the stress σ between the result

and the source data

$$\sigma = \sum_{i<j\leq n} w_{ij}(\delta_{ij} - d_{ij})^2 \tag{4.1}$$

which accumulates the residual euclidean distance (dissimilarity) errors of the low-dimensional projection d_{ij} with respect to the original high-dimensional distance δ_{ij}. The distances can be scaled by weights $w_{ij} \neq 1$ to express a quality or confidence for each pairwise distance value. The stress itself is often normalized by $\sum_{i<j\leq n} d_{ij}^2$. Standard MDS is very costly ($O(N^3)$) as it performs an eigenvector analysis of an $N \times N$ matrix and chooses the 'most significant' of dimensions, discarding the others. However, it has been improved to sub-quadratic cost over the years since the specific algorithm employed to reduce the loss function and generate low-dimensional coordinates can be freely chosen (MDS designates the general method, not a specific mathematical construct) as long as the stress is minimized, which also allows for more flexibility as the resulting coordinate system need not necessarily be an embedding of the original space. The first example of an improvement is a spring-model based algorithm with $O(N^2)$ cost [Cha96], which is obtained by calculating forces only between a point and a very small, iteratively refined set of neighbors as well as a slightly larger set of randomly chosen other points from the whole dataset. Further improvement could be achieved when borrowing the interpolation method presented in [BG98] and executing the algorithm only on a subset S of $\sqrt{N}$ points. The remaining items are then laid out relative to their nearest neighbor in S. The result is refined with a fixed number of iterations of the spring-model algorithm, resulting in an overall complexity of $O(N\sqrt{N})$, which is actually dominated by the nearest neighbor search for each point [MRC02]. The most recent improvement was made by widening this bottleneck through the use of a pivot-based quick discarding of points beyond a certain threshold [CMN01], caching a full distance matrix exclusively for those pivot points (actually only a vector remains). This resulted in a cost of $O(N^{\frac{5}{4}})$ [MC03]. To generate a force-directed layout in similarity space, simulated annealing has also been employed, which probabilistically moves the items in question to obtain a state with lower energy, reducing the temperature of the system gradually to bring it to a halt [CC92]. Other approaches include Self-Organizing Maps [LSM91] and Principal Component Analysis [WTP+95].

4.2. FastMap

Overall, these algorithms are of super-linear complexity, which makes FastMap [FL95] a very attractive alternative. This approach only has linear cost and is basically an iterative projection into low-dimensional space. Each destination dimension is created by heuristically choosing two very distant objects, which act as so-called *pivots* for this dimension. The whole data is projected

onto the axis spanned by the pivots to determine one of the resulting coordinates. Then every original point is projected onto a hyperplane perpendicular to the axis, reducing the dimensionality by one and starting the search for pivots for the next result coordinate. Since the hyperplane spans exactly those two coordinates whose details are lost with the previous projection, it is the ideal choice for recovering the error made. This means that with every additional target dimension the low-dimensional distance exhibits less deviation from the original distance, as is also proven by the low resulting stress measured in the original publication. The equation for the projected coordinates x_i is

$$x_i = \frac{{d_{ai}}^2 + {d_{ab}}^2 - {d_{bi}}^2}{2d_{ab}}, \quad a, b, i \, \epsilon \, N \tag{4.2}$$

where a and b designate the pivots for the current dimension, and d_{ij} is the distance function for the current dimension between objects i and j. To accomodate the hyperplane projection, for the first result dimension d is the original distance function in the high-dimensional space, be it a distance matrix or some measured value. For the subsequent dimensions this is changed to

$${d'_{ij}}^2 = {d_{ij}}^2 - (x_i - x_j)^2 \tag{4.3}$$

recursively, which fits the error correction from the previous projection. Note that the cost of the whole projection is highly dependent on the cost of the distance measuring, but if there is enough memory to fully cache all distance matrices, this algorithm becomes extremely fast. The highest cost stems from the pivot searching, as for each target dimension the most distant objects have to be determined. The heuristic works as follows: an object is randomly chosen and set as pivot b. Then a is set to the farthest object from b and b again to the farthest object from a. This last step is repeated a fixed number of times to stay in $O(N)$, but is very costly nonetheless.

With the availability of a well-performing dimensionality reduction method the question arises whether and how the user could tweak this mapping step to optimize its results. Depending on the occasion, the user might have additional meta-information about the dataset that he could employ to enhance the results, for example if one of the attributes is derived from (and thus dependent on) others, or if some of them are especially noisy or error-prone. So the user should be able to choose the subset of dimensions to reduce from as well as provide the distance metric since it is not trivially defined for text attributes, for example, or application-specific values that show a particular behavior.

4.3. Resulting Workflow

The proposed workflow for flexible interaction with high-dimensional source data can be seen in figure 4.1 and will now be described in more detail. The

general idea was to give the user an interface that shows the available high-dimensional attributes along with their type and range and a sample point for verification. Based on this data the user should be able to recognize any particular attributes he has additional knowledge about and thus define the input for the dimensionality reduction.

The reduction process first maps all dimensions to $[0, 1]$ for equalization, then the data is reduced to 3D. As the number of points in a dataset can also grow quite large, the visualization performance should be independent of the number of points. Considering the rising popularity of coupling scientific visualization with information visualization, volume rendering seemed to be the most obvious choice. An early step into this direction was made by [Bec97], visualizing relational data in volumes, but limited to depicting a single attribute. Rendering several attributes beyond the positional ones has already been tackled with volume rendering in the field of flow simulation [HM03]. A sketch for chemical data has also been presented [OIEE01].

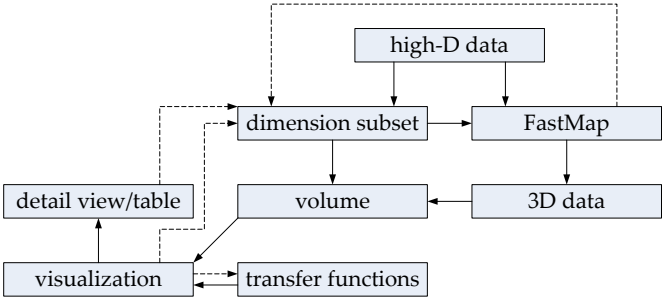


Figure 4.1.: Visualization pipeline and workflow showing the data flow as arrows and user adjustments, like filtering, as dashed arrows.

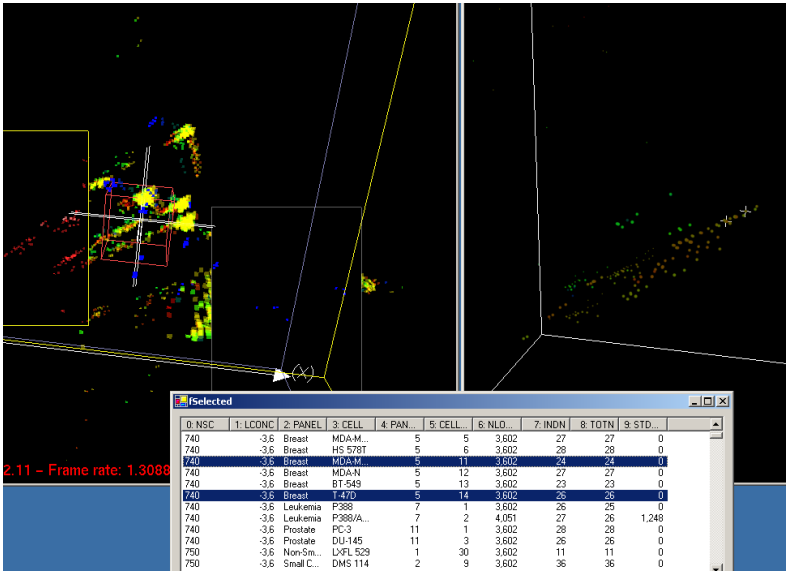
Developing those concepts further, the volume is constructed as follows: the 3D data is quantized (binned) into a 256^3 RGBA volume. The first component always contains the density, that is, the number of items that lie inside a specific voxel to allow the user to get an overall impression of the dataset structure and potential clusters of items with similar attribute values. This density can be scaled linearly or logarithmically to optionally emphasize especially 'crowded' voxels. The remaining channels can be configured by the user by choosing any available attribute to map. The user also needs to choose how the mapping is to be done since there will always be multiple points in a single voxel, so there are different statistical measures to choose from, like mean, variance, minimum or maximum. Even though the mean seems like a satisfactory default to start with, this choice highly depends on the task at hand. The points in a voxel will be reasonably similar with respect to a specific attribute

by definition, i.e. because of FastMap, but there are always categorical values which have no significant mean (resulting in wrong data being conveyed) and outlier detection obviously needs to rely on extreme values being depicted. What would also be searched for when investigating categorical attributes is the category with the highest number of representatives in a voxel, but this value is very costly to compute. The time-intensive variant would be iterating through the volume and processing all points for each voxel to find the category with most representatives (with a cost of $|V| * N$, $|V|$ being the number of voxels). The memory-intensive variant would be processing all points once and counting occurences per voxel and category, requiring $|C| * |V|$ of space, $|C|$ being the number of categories. This is still a weakness of the current tool that needs to be improved.

The volume data is uploaded to the graphics card as a 3D texture and rendered with the standard approach using view-aligned slices. The transfer functions for each attribute/color component of the data are also kept in a stack of 2D textures to optionally allow for pre-integration (see section 4.4). These transfer functions can be used to emphasize value ranges or extrema for the currently active attribute subset as well as for filtering making use of the output alpha of the transfer function. Since the transfer function editor also displays the histogram of each attribute, outliers can be directly removed or singled out, extreme values can be highlighted, and so on, without the user needing to input any specific value, as all operations are relative to the entire range for every attribute. Interactive exploration of the data is facilitated because these operations are performed in real time.

If the user has found a region in the dataset he is particularly interested in, he can highlight it with a 'volume cursor' and display the contents in a coupled view which renders all points in a 3D scatterplot, thus showing the same data but without binning it. Points can be picked to inspect all of their high-dimensional values in a table. The selection in the table is synchronized with the scatterplot and highlighting in the volume representation to allow focus + context inspection and brushing (see also figure 4.2). These interaction techniques have also been employed in a combination of scientific visualization and information visualization in previous work [DGH03, KSH03]. Based on the current results, the user can tweak the dimensionality reduction to try and isolate problems introduced by specific attributes and iteratively refine the visualization to eventually gain insight.

Virtual points can be added as a query means to the dataset at any time. This exploits one advantage of FastMap over most other dimensionality reduction algorithms, i.e. that arbitrary additional points can be projected into low-D space without altering the results for the rest of the dataset. Such a point can thus be used to search for particular attribute values since it will end up in the vicinity of its similars which can then be inspected more closely.



0: NSC	1: LCONC	2: PANEL	3: CELL	4: PAN...	5: CELL...	6: NLO...	7: INDN	8: TOTN	9: STD...
740	-3,6	Breast	MDA-M...	5	5	3,602	27	27	0
740	-3,6	Breast	HS 578T	5	6	3,602	28	28	0
740	-3,6	Breast	MDA-M...	5	11	3,602	24	24	0
740	-3,6	Breast	MDA-N	5	12	3,602	27	27	0
740	-3,6	Breast	BT-549	5	13	3,602	23	23	0
740	-3,6	Breast	T-47D	5	14	3,602	26	26	0
740	-3,6	Leukemia	P388	7	1	3,602	26	25	0
740	-3,6	Leukemia	P388/A...	7	2	4,051	27	26	1,248
740	-3,6	Prostate	PC-3	11	1	3,602	28	28	0
740	-3,6	Prostate	DU-145	11	3	3,602	26	26	0
750	-3,6	Non-Sm...	LXFL 529	1	30	3,602	11	11	0
750	-3,6	Small C...	DMS 114	2	9	3,602	36	36	0

Figure 4.2.: Picking linked between windows: The selection can be made in the detail view or in the corresponding table, the highlight is spanned across the windows.

4.4. Volume Rendering with Magic Lenses

To allow for additional filtering and locally defined queries, the volume rendering supports one transfer function per visualized attribute which can be applied to arbitrary parts of the volume interactively. They can be applied to user-defined object-space regions as well as screen-space regions following the magic lens metaphor [SFB94]. The object-space regions are assigned coarsely using volume primitives (boxes, spheres) that can be freely positioned inside the volume. Each of those primitives has an associated identifier which will be stored in an additional *tag volume* by rasterizing the primitives into it. The screen-space regions are rectangular marquees that are rasterized into a 2D mask texture and have priority over the object-space tag. Consequently the user can map the value ranges to RGBA depending on the attribute by defining one specific transfer function each. The fragment program does this as outlined in figure 4.3. First, the attribute to display is chosen based on the entry of the tag volume and the screen-space lens, prioritizing the latter. The result is used to choose the color component of the data volume as well as the z-coordinate for the transfer function lookup, the (optionally) pre-integrated

[EKE01] transfer functions being stacked along that axis in a 3D volume.

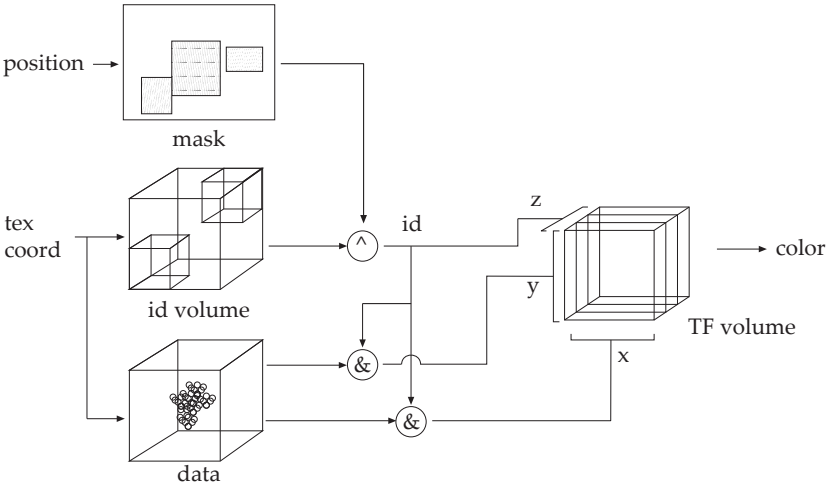


Figure 4.3.: Fragment program diagram. The attribute IDs are combined (^), prioritizing the image space mask, to select (&) one of the four volume components for each slab end. The ID also serves as coordinate for the transfer function lookup.

Categorical and string attributes require some special considerations. If interpolating in the data volume, artifacts will show whenever those two attribute types are rendered and will introduce misleading data: looking up the texture value exactly in the middle between category 1 and 3 will produce a value of 2 which might not even be present anywhere in the dataset (see also figure 4.4). In these cases it is better to use nearest neighbor and settle for one of the two, even though nearest neighbor lookup produces artifacts when the volume is zoomed in considerably. For attributes with a continuous range this does not represent a problem though, so the user can toggle this interpolation on and off.

The same restrictions as for the texture lookup apply also to pre-integration, since it replaces a linear interpolation between pairs of rendered slices. However, it considerably improves the quality of the density volume rendering if combined with interpolated lookup and can thus be activated by the user when needed. An exemplary output of such a segmented volume can be seen in figure 4.5. The tag volume segments the carp in two halves for the first and second transfer functions, the second transfer function is also associated with the lens near the carp's mouth.

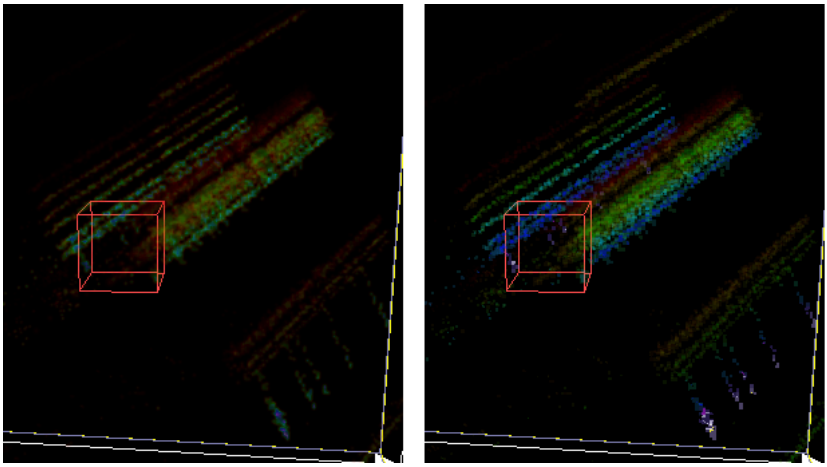


Figure 4.4.: Comparison of linear interpolation (left) and nearest neighbor (right) when displaying categorical values. On the left one can clearly see the artifacts of a different color introduced in areas of otherwise uniform color.

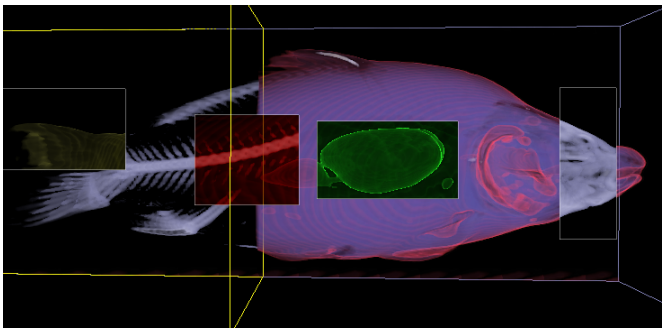


Figure 4.5.: Effect of 5 different transfer function regions. The carp is segmented into two halves using a segmentation primitive, 4 lenses apply 4 different transfer functions in image space.

4.5. Application and Results

To demonstrate the effectiveness of the proposed approach for the understanding and interaction with high-dimensional data, an example workflow and the respective results are presented. Figure 4.6 shows the `covtype` 54D dataset from the UCI Machine Learning Repository after being processed with FastMap. This results in several visually separable clusters. The dataset consists of 581,012 entries of 7 tree types located in four wilderness areas in the Roosevelt National Forest of northern Colorado. Each entry has several attributes, like a soil type classification, wilderness location, ground elevation, slope, shade, distances to hydrology etc. The four different wildernesses which were investigated form several clusters each, the shapes of which can be seen in figure 4.7 on the left, where the effect of using 4 different lenses for filtering the binary wilderness flags is collaged. If fractional distances (instead of euclidean) are used for calculating the FastMap, the measurement quality increases [AHK01] thus yielding more homogenous clusters for the wildernesses (figure 4.7 right).

From the description accompanying the dataset one can retrieve the information that the wilderness with the highest mean elevation is Neota. This can easily be reconstructed with a transfer function for the elevation attribute by setting the output alpha to 0 for the range [0,0.85], thus showing only trees in the top 15% of the maximum altitude (see figure 4.6 right). Even more

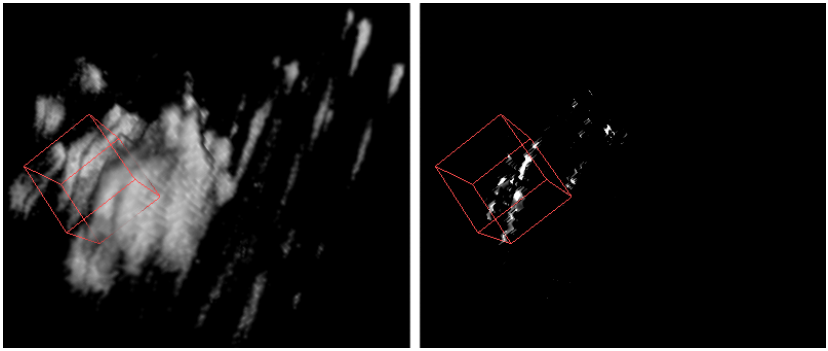


Figure 4.6.: Before (left) and after (right) filtering the trees having an elevation lower than 85% of the elevation range.

interesting data mining possibilities arise with a slightly modified fragment program which allows for the modulation of a transfer function assigned by using a segmentation volume brush with a filtering lens. When processing the volume associated with the first transfer function in such a way that each of the different trees gets a distinct color (then averaged per voxel), the result

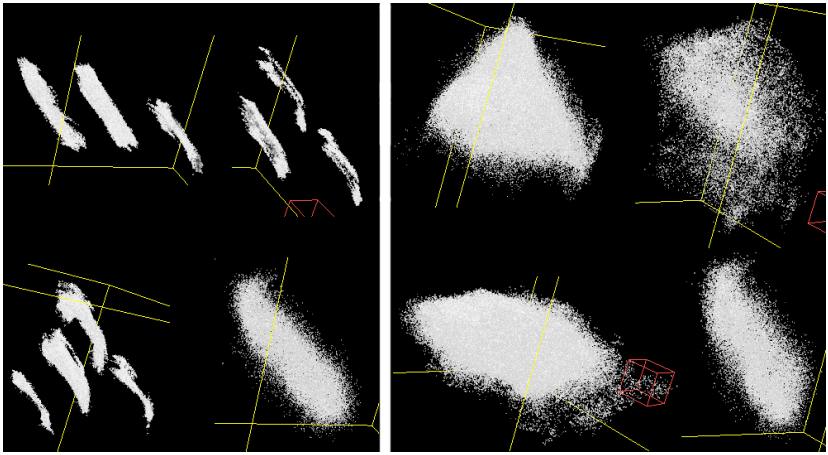


Figure 4.7.: Collage of 4 wilderness clusters, FastMap with euclidean distances to the left (higher fragmentation), fractional distances ($f = 0.3$) to the right (better defined clusters).

looks as in figure 4.8. Assigning a second transfer function filtering out low elevations lets the user visually discover the most common trees for high elevations (see figure 4.9). Another example would be to only display the tree population for a certain wilderness, in this case of Rawah (see figure 4.10). It must always be kept in mind, however, that these filtering tools work only on a per-voxel basis, so the results depend on the statistical measure used. To get all attributes for the selection, the detail view has to be used (figure 4.11) to see exactly which kinds of trees live at the higher elevations; in this case it would be exclusively Krummholz. However this kind of data mining process only makes sense since the user can apply filtering and highlighting directly to the volume view, otherwise the user would have to inspect every possible subvolume of the data to find regions of interest.

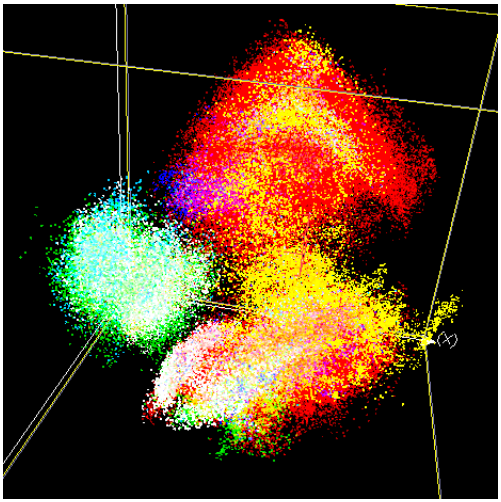


Figure 4.8.: Trees colored by type with a transfer function divided in 7 parts.

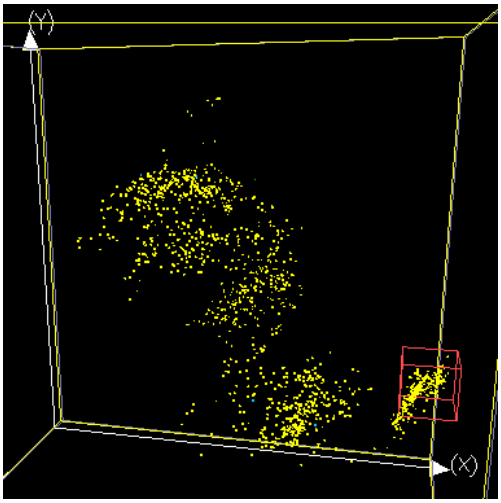


Figure 4.9.: Trees colored by type, filtered by a second transfer function making all trees beneath 85% elevation transparent.

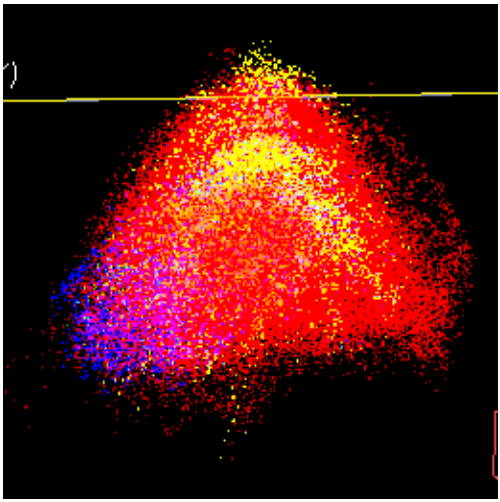


Figure 4.10.: Showing the trees of wilderness Rawah. Hue is produced by one transfer function on the tree type attribute and modulated by the transfer function filtering out the wilderness.

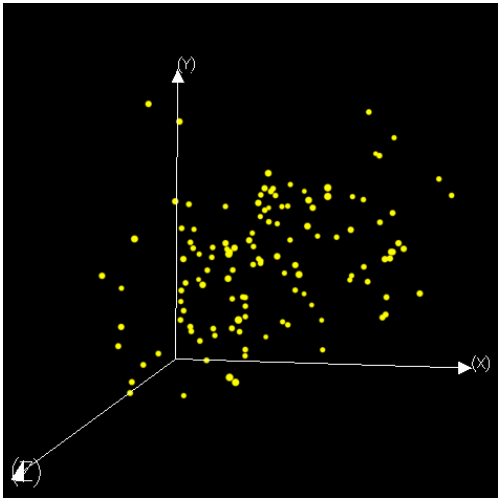


Figure 4.11.: A detail view of the actual trees in regions of high elevation, coloring taken from the transfer function in figure 4.9.

4.6. Performance

As with all volume rendering approaches, the performance of the proposed approach is limited by the fill rate, i.e. memory bandwidth of the graphics card used. Furthermore, at the time of publication of these results (2004), available graphics cards were limited to four texture lookups per rendering pass, whereas five are needed (slab front, slab rear, lens mask, id volume and transfer function, the last one dependent on all previous ones and a number of calculations). Depending on the viewport size, on a Radeon 9700 Pro, this still yielded an acceptable average of 7.5fps in a 600^2 window (averaged over all viewing angles, since 3D texture organization in ATI hardware causes slowdowns when inspecting the volume from the 'rear'). Already with a mainstream GeForce 6800 the performance increased to just below 30fps, which is more than adequate for interactive exploration[1]. The execution of FastMap on 580K points in 54D takes about 132 seconds on an Intel P4-2.8Ghz Machine, while 2.3M points in 9D take 248 seconds, to name two examples. The updating of the additional dimensions including texture upload takes about 3 and 5 seconds (without particularly optimized algorithms), so it is safe to say that the user can experiment interactively with the prototypical implementation. The performance of the scatterplot is not critical, since only a small portion of the available data has to be rendered at any one time, which was always faster than the volume rendering during testing.

4.7. Accelerating FastMap

Even though linear in complexity, for large datasets the FastMap algorithm still requires a noticeable time to complete. It is very important though to have the shortest time possible between first obtaining a dataset and getting an impression of its overall structure, concerning the distribution and similarities in it, as a first impression can already outline problems and thus save preprocessing times on datasets that turn out to be wrongly acquired or otherwise compromised. Since for the low-dimensional representation all resulting coordinates are calculated independently from each other, a SIMD approach nearly imposes itself, thus motivating the implementation on a GPU.

The utilization of GPUs as co-processors for algorithms that do not directly generate images is becoming ever more wide-spread, so a specific SIGGRAPH workshop has been created to deal with the challenges that arise [gpg]. The architecture of graphics cards is not yet as flexible as the general-purpose CPU, therefore some limitations have to be worked around before employing the graphics hardware for general calculations. Different methods have been de-

[1] Note that the GeForce 6800 was available about one and a half years later and is two generations newer.

vised for storing complex data structures in textures, and tricks have been resorted to for emulating program flow control when it still was not available in shaders, for example using the z-test [KW03a]. Support for program flow control has been introduced with the so-called *Shader Model 3.0*. Most GPGPU implementations share the property of mainly making use of the *fragment units* despite the *vertex units* being programmable as well. This is motivated by the fact that before the advent of a unified shader model (as present in the GeForce 8 series, for example) the fragment processor had more parallel-working VPUs (vector processing units) than the vertex processor, usually about twice as many or even more. Furthermore the instruction set for fragment programs was more powerful and the access to data (in form of textures) was much faster, even if not exclusive anymore since Shader Model 3.0.

This context can be taken advantage of to focus on the implications when using the GPU in such circumstances as well as providing a performance analysis to differentiate and show the strongly varying performance of some GPUs and, above all, identify the bottlenecks and computational precision issues that come into play when 'outsourcing' CPU work to the graphics card (see section 4.10). The acceleration of FastMap also offers an opportunity to study how the three-dimensional structure of a dataset is affected when changing the parameters (pivots) for the projection.

4.8. The GPU implementation

To execute FastMap on the GPU, the pivots are first prepared on the CPU by using the heuristic described in section 4.2. The source data is then split into several floating-point textures as follows: all integer and floating-point attributes of the source data are stored in groups of four in the red, green, blue and alpha channel of textures, all on the same coordinate for one single data point. Strings are mapped to unique IDs per attribute (so if there are 5 string-type attributes in a dataset, 5 distinct string lookup tables result). On the one hand this saves memory if the strings are categorical attributes and thus repeated several times in one dataset. On the other hand this makes string comparison much faster since the difference between two strings can be defined as simple inequality, and for categorical values an editing distance would not make sense anyway. The resulting IDs are also stored in texture color channels, such that all of the attributes can be accessed by using a single texture coordinate.

Depending on the number of attributes and points in the source data, the texture and stack size is varied, with differing impact on rendering performance (see section 4.10). Only one texture stack is processed per rendering pass, yielding one projected coordinate for $texture\_size^2$ points. This rendering process is repeated until all input data has been reduced to lower dimensionality (resulting in $\lceil \frac{n}{texture\_size^2} \rceil$ iterations). The stack of textures is complemented

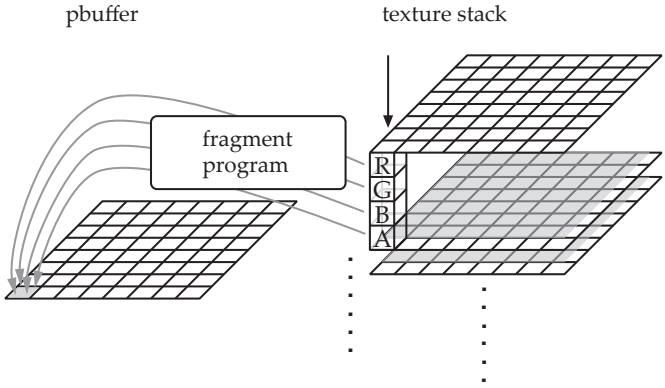


Figure 4.12.: Data as textures on the graphics card; one 'column' contains all attributes of one data point.

by another floating point texture, a *pbuffer*[pbu], as the rendering target with the same resolution as the texture stack. The x_i (see equation 4.2) calculated from the input stack is stored inside this pbuffer. If the pbuffer is bound as texture, the results of a previous projection can be easily accessed and used for the calculation of the modified distances d'. All result dimensions are computed consecutively for each stack, in order to keep the source data textures untouched for two additional passes (in this case of a 3D similarity space).

To trigger the calculations, a single quad covering the whole viewport is rendered to generate the texture coordinates for processing every data point/pixel in the input texture stack. The viewport has the same dimensions as the pbuffer. An *ARB_fragment_program* is used that stores an x_i as above in the pbuffer, depending on the distance calculated from the source data and the attribute values of the pivot points, which are passed as program parameters. This is justified by the fact that the pivot point could be in another texture stack and thus would not be accessible. The pivots could also be stored at a constant position in the input stack, which would reduce the data processed per iteration by two items, but the main advantage is that $2 * stack\_height$ texture lookups can be saved per result fragment if program parameters are used. The pivot points are constant for the whole stack and one rendering step, in any case. $d_{a,b}$ and ${d_{a,b}}^2$ from equation 4.2 are also constant and therefore passed as parameters as well.

The fragment program basically consists of three blocks of code: in the first block the floating point attributes are retrieved, subtracted from the pivot attributes, squared and added up, yielding a quadratic euclidean distance. String IDs are checked for equality against the pivot values and added accord-

ingly in a subsequent block. The last block calculates x_i. Using the square root operation can be completely avoided since all the distances calculated before are squared again in this block (see equation 4.2). The necessary code is generated dynamically based on the dimensionality and dimension types of the input file; only the source texture unit and the program parameter containing the respective attributes of the pivots must be set accordingly.

The fragment program for subsequent dimensions is generated in the same way and just needs two more parameters (the latest projections of the pivots) and one more texture fetch for the latest projection of each point itself (from the pbuffer of the preceding pass), along with another code block for the calculation of d' from d using these latest projections. After enough dimensions have been calculated (up to four per pbuffer), either the results can be read back to the main memory and used as a vertex array to display the resulting low-dimensional representation of the data set, or the pbuffer is directly used as a Vertex Buffer Object. At the time of publication also a superbuffer could have been used, but only ATI cards supported it and practical tests performed by colleagues confirmed that it did not work very reliably.

4.9. Application

The implementation has been tested with different excerpts from a 10D dataset of 2.3M points retrieved from a cancer screening database projected into 3D similarity space. This dataset contains 8 numerical and 2 categorical alphanumerical attributes and quantizes the reaction of various cancer cells towards different substances in terms of cancer growth inhibition. In figure 4.13 the result from using the original heuristic on a 1M point subset is shown in the application. The OpenGL window depicts the dataset in 3D similarity space with the pivots highlighted and colored according to the target dimension they belong to (in RGB order). A short fragment program is used to cool/warm-color-code the scatterplot depending on the z depth of a point. Since unshaded points do not occlude each other in a way that allows for satisfactory visual depth perception, this approach was used to convey a better impression of depth to the user [WE02], see figure 4.14. Intensity attenuation would also have been an option, however the darkened points would be quite difficult to perceive because of their small size. The parameter window allows the user to scroll through the different pivot points in real time or even to animate one of the axes with adjustable step size and delay between steps. figure 4.13 shows some well-defined streaks with similar data, however the result points all lie on a plane, so only two of the available three dimensions are used.
When tweaking the pivot points by hand in real time, other cluster-like arrangements can be spotted. However it has also been found that, despite the original heuristic, choosing pivots that are quite close on the resulting axis can yield projections which are fanned out more thoroughly than when always us-

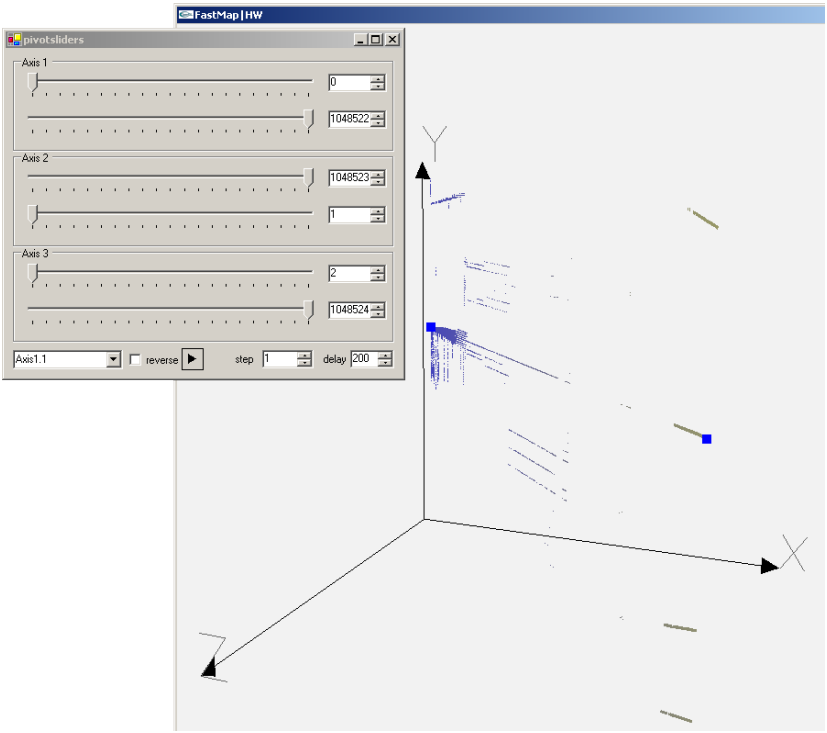


Figure 4.13.: Application user interface with OpenGL 3D scatterplot (right) and FastMap parameter window (left). Pivots have been chosen using the original heuristic.

ing points that are as far from each other as possible. Figure 4.14 shows such a hand-tweaked result and suggests that three major clusters exist, the bottom one being clearly composed of overlapping streaks, which hints at several series of data with slowly varying attributes. This does not mean that the original projection is faulty or useless, but that different characteristics can be spotted in one dataset when making use of human interaction and experimentation as an added heuristic to complement the automated one.

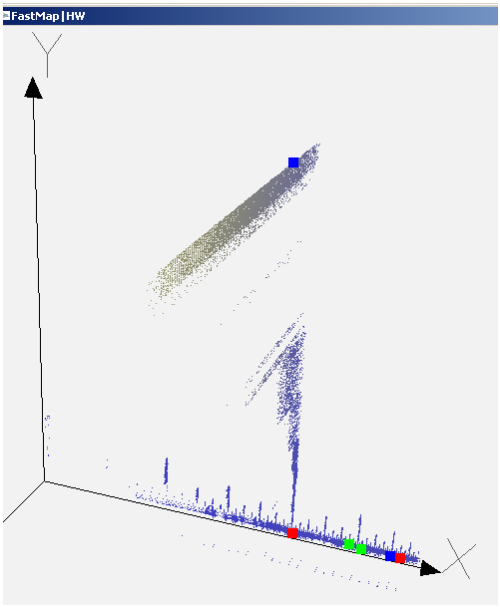


Figure 4.14.: FastMap result with tweaked pivots. The result points are much more widely spread in 3D. Cool/warm shading is employed to emphasize the point depth.

4.10. Performance Discussion

To demonstrate that the proposed approach is an improvement over CPU-based FastMap, timings of the implementation have been taken for various datasets on an Intel Pentium4 running at 2.4 Ghz and on a GeForce 6800GT (see figure 4.15). The CPU-based FastMap also uses only 32bit floats in order not to penalize its performance further by using doubles. Different excerpts from the cancer screening dataset mentioned in section 4.9 were used. The first subset is very small (26824 items) while the second consists of one million items. The higher-dimensional datasets are just repetitions of the original data to allow for simple investigation of the performance variation when increasing texture reads per result point. One can see that the GPU implementation clearly outperforms the CPU variant by orders of magnitude, allowing for interactive adjustment of the pivot points. The projection times stay below one second for source data sizes of up to two million data points and 10 dimensions (since two color channels are left in one of the source textures, about the same order of performance is valid for up to 12 dimensions).

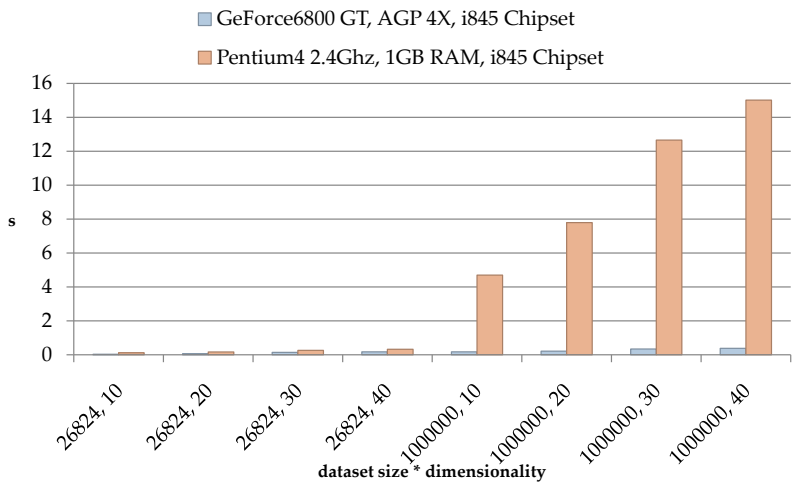


Figure 4.15.: Total time in seconds for FastMap on CPU and GPU with optimum texture size

To further investigate how the different parameters of the implementation, like texture sizes, system bus types, and, last but not least, graphics chipsets, affect these numbers, more tests were conducted. One would think that the optimal parameters for executing such an algorithm in graphics hardware would be the use of as few and as large textures as possible, to keep management overhead at a minimum. The textures would be created once and reused for every stack that has to be iterated. By and large this is true, but the measurements taken show where specific strengths lie for different GPUs and pointed out some unexpected flaws that must be taken into consideration when using them.

The GPU FastMap is implemented in OpenGL in order to retain the option of easily integrating it with existing OpenGL-based scientific software which could benefit from the fast dimensionality reduction. This poses a challenge when it comes to performance measurements which go beyond simple FPS for final visualization. Therefore the measurements were repeated many times and it was also made sure that the GL pipeline was flushed after the timed phases. The measurements were also taken with constantly varying pivots so the constantly changing output could also be visually verified to make sure the graphics drivers would honor the *glFinish()* request. One can see in figure 4.16 that the overhead for constantly replacing the contents of small texture stacks is much higher on Nvidia chips than on the ATI chips, so if the whole graphics

card memory cannot be used for FastMap, the approach becomes extremely slow. However with big textures the results are similar to the ATI X800 AGP.

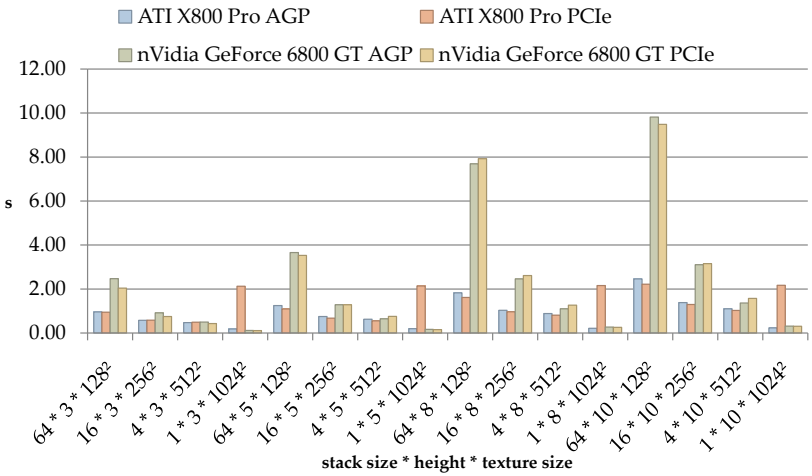


Figure 4.16.: Total time in seconds for FastMap on different GPUs with different tiling of the 1M dataset and increasing dimensionality.

Another irregularity discovered was extremely bad readback performance on the PCIe X800 card with texture sizes of 512^2 and above, despite the acceptable results from the AGP variant. Comparing the calculation of the 1M 10D dataset, moving from four stacks of 512^2 textures to one stack of 1024^2 decreases the performance by four times, after the already irregularly small increase from 256^2 (see also table 4.1). Also a side-effect was encountered on an ATI X800 Pro AGP, where the 20-dimensional dataset with 1 million items distributed over 4 stacks of 5×512^2-sized textures produced a delay after each calculation, so the measured 623ms produced only one result frame per second. This seems to be a driver bug, since for example 4 stacks of 8×512^2-sized textures resulted in no such delay, and the same executable with the same parameters did not cause a delay on any other configuration. This practical example also serves very well as an illustration that different drivers or different code paths in the same unified driver can have various side-effects which makes benchmarking overall a very difficult task as it costs much too much time to manually try out enough different hardware-driver combinations to be sure to get relevant results.

Readback performance has been cross-checked with an individual test, the results of which can be seen in table 4.1. It is conspicuous that the readback

card	512^2	1024^2
GeForce 6800GT AGP	750.47	751.31
GeForce 6800GT PCIe	818.92	829.37
ATI X800 Pro AGP	117.21	116.36
ATI X800 Pro PCIe	235.32	7.56
ATI 9700 Pro AGP	114.03	113.31

Table 4.1.: Readback performance in MB/s for RGBA float pbuffers of given size from a particular graphics card.

performance has not improved moving from ATI 9700 to X800 (for the AGP cards). The result for 512^2 textures shows that the native PCI Express interface on the X800 cards, on the other hand, is an improvement over the AGP interface, however the cards cannot catch up with the current Nvidia chips (at least when working with float formats). The lack of a significant performance improvement on the GeForce when moving from AGP to PCIe is likely caused by the fact that the chip does not have a native PCIe interface, but uses a transponder chip instead.

An additional important factor for the overall performance of the algorithm is the execution speed of the code on the GPU. A detailed analysis has been conducted in this regard [DE04], however the specific effects on this particular case need to be investigated. In figure 4.17 one can see that the ATI chips have a big advantage over those on Nvidia cards. The cause of this is two-fold: the ATI cards have a much higher core clock and make use of only 24bit-floating-point VPUs (see also section 4.11). It is also obvious that the ATI X800 scales much better with increasing number of texture reads and constant texture size than the GeForce 6800, but has a weak spot when it comes to 512^2 texture sizes. The ATI X800 PCIe could have the performance lead (if not the precision lead) were it not for the much better readback performance of the Nvidia cards regardless of the system bus employed. The CPU bus and chipset should also not be underestimated for its performance impact. Comparing an i845-based system with 400Mhz FSB and AGP 4X to an i865-based system with 800Mhz FSB and AGP 8X, using a GeForce 6800GT in both cases, a performance increase of over 200% was measured instead of the optimistically anticipated 100% which demonstrates that also different chipsets and the increased FSB bandwidth have a significant performance impact.

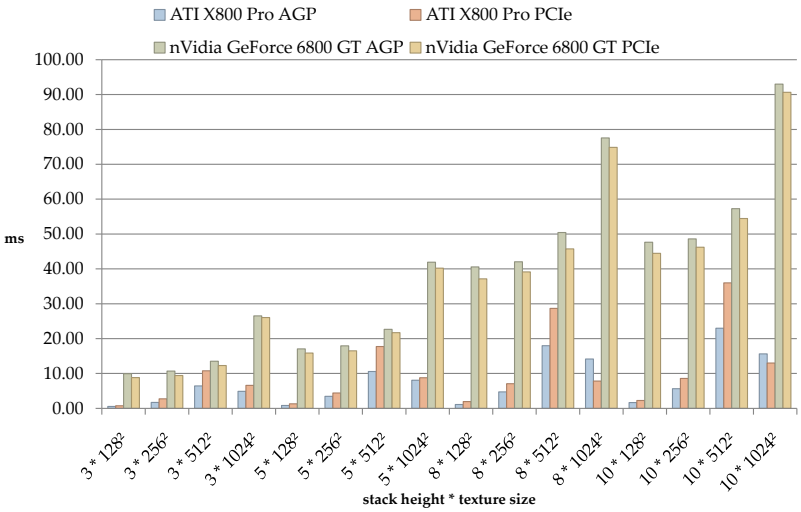


Figure 4.17.: Fragment program execution time in milliseconds per stack with different stack sizes and texture sizes. ATI chips show a clear advantage over Nvidia chips.

4.11. Precision Issues

If the GPU is used to replace the CPU, a major issue that arises is the computational accuracy. The surveyed ATI chips offer only 24 bits for computational accuracy, even though source and result data is stored as 32bit IEEE floats. Nvidia chips allow the programmer to select between half (16bit) and full (32bit) precision floats for computation, while storage relies on 32bit IEEE floats as well. One advantage of the CPU over the GPU is the availability of higher-accuracy number formats and the possibility of at least using these for calculation, even if the results are then stored only as floats (to save memory, while still keeping the accumulating error at a minimum). The effects the precision has on the relative error of some example operations can be seen in table 4.2. These values are obtained as follows: A small test program uploads a RGBA float texture to the GPU, activates a fragment program and draws a viewport-filling quad into a pbuffer. Then the pbuffer is read back and compared to CPU-based results calculated with double accuracy which represent the ideal, if not correct, result. This obviously is a worst-case scenario since also on the CPU normal floats would be less precise. The texture contains $512^2 * 4$ floats with the value $\frac{1}{512}(index + 1)$, that is, values in the range $[\frac{1}{512}, 2048]$. As a first test the values were simply passed from the texture to the pbuffer (x

Operations	ATI X800/9700	Nvidia GF6800
x	0.000003636	0
$x * x$	0.013201884	0.000023515
$x\frac{1}{x+1}$	0.000016217	0.000000005
$(x+x)(x+x)$	0.052807537	0.000094060

Table 4.2.: Computational accuracy (average relative error) for selected operations on different GPUs.

in the table). Since the data has to pass through the VPU, this already causes some error on ATI chips. The other tests execute some simple calculations. A different denominator from x was chosen in the reciprocal multiplication to avoid that the operation simply be discarded by optimization in the drivers. It can be seen that the 24bit VPU accumulates a large error quickly, however the implications only become clear when considering some applications. Calculating a position for displaying data (as is also the case with FastMap) and normalizing each dimension to allow the algorithm to fill a unit cube, and then displaying this cube at screen size (1200 pixels in height), this would result in an average vertical positional error of 60 pixels on an ATI card when considering the 5% relative error ($(2x)^2$ in table 4.2). On an Nvidia card this error would amount to barely $\frac{1}{10}$ of a pixel, which is more than sufficient. As these cards are originally intended for delivering good performance and visual effects for computer games, this can be considered adequate, but for general processing on graphics cards it is a factor that must be kept in mind (especially when using ATI cards).

The error of the FastMap algorithm was measured as well and compared to FastMap on the CPU (using only floats as in the performance comparison). For the 1M dataset the ATI card produced an average relative error of 0.000294823, the Nvidia card one of 0.000056497, so the discrepancy is well below one pixel in both cases if the same assumptions as above apply.

4.12. Summary

This chapter proposed a tool and the related workflow for the overall analysis of large, high-dimensional datasets. The user can, in an iterative process, map the data to 3D and then analyze its structure using a volume representation of the resulting similarity space. Additional dimensions can be displayed in the volume by user interaction for highlighting or filtering in real-time, and the user can then drill down and more closely inspect the areas of most interest in a coupled detail view as well as in a table with the exact value, all of them linked to allow for brushing. This approach works well for large datasets, which has

been proven by showing the performance and results using a dataset with 580K entries. To further improve the performance of this approach, graphics hardware has been employed, opening up the possibility of interactively changing the parameters and observing the structural changes in the resulting low-dimensional data. This improvement also offered itself as a background to discuss the more generally applicable performance and accuracy constraints that come into play when using GPUs as co-processors, pointing out some pitfalls to keep in mind when considering to move an algorithm from the CPU to the GPU. From this example one can see that if we keep these limitations in mind, a modern GPU can provide a very cost-effective way to execute massively parallel algorithms. With the availability of high-level languages like GLSL or Cg, or specialized frameworks like Brook [Buc03], [BFH+04] or Sh [MQP02], the porting of an algorithm has become quite easy if one finds an efficient way to map the application's data structures onto 'flat' arrays (textures), even more so with the new GPGPU APIs CUDA [CUD] and OpenCL [Opea].

5 User Interaction

The approaches described in the preceding chapters help realizing the visualization pipeline as introduced in chapter 2. What is still missing is part of the feedback loop that allows the user to steer the way this pipeline works, especially the filtering and rendering stages, i.e. choosing the subset of data that has to be rendered, inspecting particular items, and navigating the 3D representation. This chapter will discuss a special case, that is user interaction in an immersive environment, meaning ways of interacting with a stereo-enabled representation that allows for depth perception without having to handle conventional, uncomfortable-to-carry input devices like mice and keyboards and above all not interfering with the stereo output, which increases the perceptual stress considerably. For a study on the subjective perception of immersion, refer to [Hee92], a taxonomy for immersion has been developed by Robinett [Rob92]. This scenario was chosen because more often than not the datasets the presented visualization approaches are tailored for contain great numbers of quite small items that by itself give very weak depth cues as the individual forms are too small to occlude each other significantly which does not give the eye enough information to guess which one is in the foreground and which one behind. Thus, to convey the spatial data structure more clearly, depth perception by disparity, as offered by stereo displays, helps significantly.

Even when the performance of the visualization system is sufficient to handle the large datasets as output by simulations, for millions of data points all individual characteristics are difficult to grasp by the human researcher. An overall structure will be intelligible, but the more attributes and details are shown, the more convenient it is for a scientist to concentrate only on a smaller number of items. To overcome this problem known as *visual overload*, several approaches can be considered, for example a sensible abstraction from the single data points (see section 3.2.2) or filtering. Another feature needed alongside the visualization is picking, which is actually a special case of filtering, easier to undo and often resulting in smaller subsets of the data. It enables the user to inspect attributes that either cannot be efficiently visualized because

The work in this chapter is based on [RRE06] and some paragraphs and figures are excerpted from the original publication.

of their type, or because there simply are too many attributes to be shown simultaneously, by giving selected items a special representation or additional information displays.

The other aspect of user interaction covered in this chapter concerns the navigation of the dataset, more specifically the manipulation of the position and orientation of the dataset relative to the observer, which is depicted in the last not yet covered feedback connection in the visualization pipeline.

5.1. Related Work

The scenario for the required interaction consists of a relatively large space filled with tens of thousands, and up to millions, relatively small and in parts extremely densely packed particles. Usually there is floating matter with low density in a large part of the space and a smaller number of, in some way, coagulated particles, or clusters (droplets, galaxies, etc. depending on the simulation domain). The main requirement is that the users have tools which allow them to single out clusters or other interesting parts of the dataset as well as being able to pick single particles, if needed. The hardest problem in this scenario is clutter, which is even worse for the original galaxy simulation visualization because the particles are blended and thus the user cannot easily distinguish overlapping objects. This adds uncertainty when it comes to selection because the user first has to find out which of these objects in a certain region he is really interested in. A comprehensive taxonomy on selection mechanisms and operations has been presented in [Wil96], which has been the base for deciding the features for the VR user interface. Mouse-and-keyboard-based interaction is probably the most common paradigm overall for workstation scenarios, however in VR and AR environments there has been considerable research not only about methods but also about more suitable input devices.

The basic method of pointing-and-clicking has the benefits of being widely known and accepted. There is next to no *Gulf of execution* (the user wondering what he might be able to do in a particular situation [Nor88]) since most users know by convention that clicking the left mouse button will result in the selection/activation of whatever the cursor is hovering over. The results are precise and it is very easy to switch between several operations quickly and easily using different buttons and hotkeys (which however will have to be learned by the user). The corresponding approach for immersive environments is the laser pointer and its many improvements like the spot light [LG94] or IntenSelect [dHKP05]. These approaches try to compensate the human motorical deficiencies by adding fuzzyness and/or a ranking method to decide which of the possible candidates the user really wanted to select. Such improvements take into consideration small and far-away objects as well as clutter, however comparing the datasets used for testing to the simulation scenarios that are de-

scribed above, one will note the several orders of magnitude higher number of objects in a scene. Other common metaphors for VR/AR interaction have some correspondence in the real world or are inspired by it, for example, pen and tablet [BBMP97] or variations like pointer and clipping plane [dRFK+05] and real hand-held devices [WDC99]. An example for a virtual device controlled by a completely different commodity peripheral is the virtual tricorder/space mouse combination [WG95].

5.2. Dataset Types and Scenarios

The application focus in this context was chiefly on two specific fields of research. One is multibody simulation from astrophysics, more specifically the simulations carried out by the VIRGO supercomputing consortium, which yield point-based datasets with several millions of particles. These are visualized with the point cloud renderer the molecular visualization is based upon [HE03], however to improve performance even further and reduce visual overload the option of interactively trimming the dataset would make it easier to focus on the interesting parts of the data. For this purpose the different selection operations are very important to allow a comfortable drilling down into smaller areas. The other application field is thermodynamics, where datasets are smaller (ten thousand to millions of molecules), but the focus is on inspecting the attributes of particles (for example to verify cluster detection or the simulation algorithm) and slicing clusters of particles. In this area researchers are just starting to make use of the performance boost commodity clusters are offering, so dataset size is not a primary problem for now.

Since the basic framework optionally renders the data adaptively, and tries to maintain a frame rate of at least 10 fps, any filtering of the dataset results in the framework having more time to traverse and render the remaining points. Therefore such filtering effectively causes a refinement of the remaining regions if parts of the data had to be skipped as long as the whole dataset was visible, increasing the visual quality considerably.

5.3. Features and Implementation

The user interface devised has two principal modes, one for workstation use in conjunction with standard PC hardware and one for use in VR environments with stereo output and tracked interaction. The whole system is based on the concept of a two-tiered selection consisting of a *temporary selection* and a *permanent selection*. This allows the user to create complex selection subspaces incrementally using the temporary set and to store satisfactory results in the permanent set without the risk of compromising the permanent selection with the next selection interaction. The storing of temporary sets is effected through

boolean operations, so the user can add to, subtract from, or even intersect the temporary selection with the permanent selection. Other operations that can be performed on the data are:

- cropping to selection sets
- hiding of the temporary selection for incremental refinement – the permanent selection cannot be hidden because it represents the result of all the filtering operations and thus must not disappear
- single-click toggling the hidden data to enable the user to re-check the context of the selected data without losing the selection subset.

The user interface offers an additional window which contains the interaction menu and a table (see figure 5.1). This table displays the attributes of all particles contained in the permanent selection, so the user can look up the exact values as they are output by the simulation, which is particularly helpful for particles that have more attributes than can readily be displayed by the available rendering modes, but are important for the understanding of the state of the simulation. The 3D view of the dataset is linked to the table, so the user can select subsets of particles in the table which is reflected by bracketing the selected points in the 3D view by one bounding rectangle each (brushing).

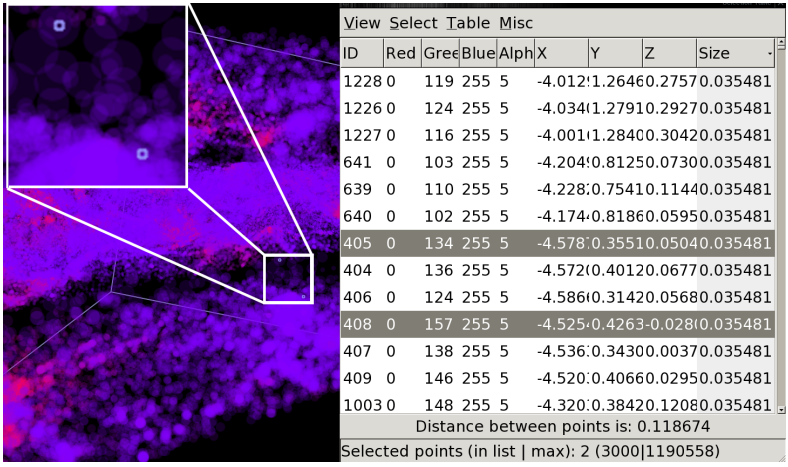


Figure 5.1.: Selection in the table and linked points in the 3D view, highlighted by a small bounding box.

Additionally, the user can interactively adjust the near and far clipping planes of the GL context to coarsely select the spatial subvolume of the dataset he wants to interact with. Such cuboid 'slabs of interest' give the user another basic volumetric primitive by which he can specify the focus region he wants

to interact with. An interesting slab can be "accepted" – marking all invisible points as invisible and resetting the planes to their default position for further interaction. The near and far planes' settings are visualized by rendering semi-transparent rectangles at the modified near and far planes. The rectangles have the size of the visible part of the old near plane.

5.3.1. Workstation Interaction

When using keyboard and mouse for interaction, the keyboard can be used to gain access to almost all the functionality very quickly by using hotkeys. Less advanced users may choose to use the popup menu or the regular menu for triggering the interaction modes. The actual selection is always triggered by using the mouse. The user can drag a rectangular selection marquee as well as paint it with a brush of variable size (as known from common graphics applications). Both selection types are projected infinitely in depth based on the current view, so the result is always a frustum with an arbitrary profile. The depth is not limited in this case because the astrophysics datasets have high overdraw and use blending, which makes it difficult to decide at which depth a point lives without rotating a dataset when no stereo output is available. The depth filtering can be easily performed by modifying the resulting frusta from different points of view. These selection actions always create and modify the temporary selection which can later be baked into the permanent selection. A virtual sphere (trackball) metaphor is used to navigate in the dataset.

5.3.2. Tracked Interaction

This mode is implemented making use of the optical tracking system already installed at our institute, which consists of four ARTtrack1 cameras. This infrared-based optical tracking system is capable of tracking the position and orientation of multiple targets, or *bodies*, which consist of several markers, and are configured to be worn on glasses or hands. Since these targets do not support any spatial reconfiguration (as they are recognized by the spatial distribution of markers), gestures or finger movements cannot be distinguished. To have some kind of triggering option, a so-called *flystick* (see figure 5.2) was used, which is basically a marker-equipped joystick handle with several buttons and radio-based communication with the tracking server.

When using the flystick, the main difference to mouse/keyboard interaction is the visible 3D cursor (a 3D arrow). The user can press a button on the flystick to make the current cursor position a pivot point around which the dataset can be rotated or in relation to which the dataset can be translated. To decouple the selection and navigation precision from physiological precision of the users' movements and spare the user the frequent repositioning of himself in front of the powerwall, the cursor movement is not necessarily mapped absolutely

Figure 5.2.: The flystick used for interaction has several buttons for triggering interactions and reflecting markers for tracking with IR cameras.

from the flystick movement in front of the powerwall. Instead, the user can fixate the 3D cursor by pressing a button, and then reposition the flystick to a more comfortable position, thus emulating what for a desktop user is the lifting and repositioning of the mouse.

The position of the 3D cursor is visualized by a half-transparent sphere in the background and a real "cursor" in the foreground. Additionally, when the cursor intersects with points, these are colored differently to ease their spatial localization. In this mode, the brush selection is not projected, since the user can position the cursor in all three dimensions by direct interaction. Thus all the highlighted points are added to the temporary selection when the corresponding button on the flystick is pressed.

The table with its detailed display of particle attributes is also available in VR mode and projected onto the focus plane. However, since it is quite difficult to accurately grab the scrollbar or the corresponding buttons when interacting with the flystick and thus resorting to one's gross motor skills, the list of selected particles is scrolled differently: to allow easy relative and also absolute scrolling, a modifier key and pointing direction are used. When pointing into the list window, scrolling is relative, outside to the right (where the visible scrollbar lies) scrolling is absolute (and actually more of a 'jump to' functionality). Scrolling itself works by pointing up or down relatively to a "virtual horizon" in the middle of the screen. Simple selection within the table is possible through pointing as well and "Shift" and "Ctrl" are mapped to flystick buttons to allow for the selection of multiple items.

5.3.3. A 3D Candle

In this mode, the cursor disappears and a halo of the same size as the 3D brush is used to select points for rendering, as if the only source of light in the visualization were at the position of the cursor and everything beyond the halo were submerged in the dark. That way, all the processing power can be used for a small region of the dataset, which ensures maximum detail in the focus region while completely neglecting the context. This metaphor was chosen for two reasons: on the one hand, it allows for extremely high refinement and high performance since only a very limited part of the dataset has to be rendered, and on the other hand it reduces the high overdraw and the resulting visual overload for the user drastically, especially in snapshots from cosmological simulations with several tens of millions of particles. Since this is only a metaphor, the lighting conditions of the rendered scene were not changed at all because (at least in case of the molecule rendering) back-lit primitives offer very low contrast and no sense of spatial extent. The metaphor has the additional benefit of being easy to understand if seen in context of the real world (where the effect is borrowed from in the first place).

5.3.4. Measures to Avoid Fatigue

A combination between pointing to the powerwall (for the selection list) and a directly mapped 3D cursor is used to faciliate interaction. In a first attempt to provide a mode selection in this virtual reality scenario, a popup menu on the focus plane was used for user input. This proved to be unsuitable as mapping the flystick interaction to a relative mouse pointer movement on the focus plane was precise, but caused tremendous fatigue, causing the users' hands to slightly drift to the lower right after some menu interactions. Although it would be easily possible to recalibrate the hand's position to a less exertive position, this feature would need to rely on some kind of fatigue-perception algorithm. In a second attempt, the mouse cursor was positioned at the intersection point between a virtual ray cast along the flystick's z-axis and the focus plane. This removed the fatigue source as the user automatically relaxed his hand position when switching from the relative cursor movement manner to the pointing approach. However, this solution caused a loss of precision stemming from the combination of motor precision and tracking precision – small angle changes when standing two meters from a target cause considerable positional imprecision. The accidental selection of the wrong menu item was the most frequent result. To solve this problem, larger menu elements and a menu type that fits the directional pointing method far better were used: the radial/pie menu. It is by design superior to the linear one in terms of target size and cursor travel distance ([CHWS88], see also Fitts's law [CNM83]).

5.3.5. The Radial Menu

The classical radial menu, which can be seen in applications with a mouse interface, e.g. Maya, was extended to a full-screen approach. Thus, the whole display area is used as a projection target allowing menu items to be significantly larger in both dimensions. All menu items are shown at once. The amount of information may seem very large, however by highlighting only parts of the hierarchical structure, the user can concentrate on suitable amounts of information: only 5-9 sub menu items are highlighted, just few enough to not unduly overload short-term memory, which is limited to about 7 *chunks* [Mil56]. All menu items are displayed as icons with a tooltip showing a short description of the item at the cursor's position when hovering over it. All submenu entries are only outlined by default, the corresponding icons are displayed when the corresponding menu entry is hovered over (see figure 5.3). Another advantage of this approach is that the user can more easily memorize the position of menu entries, and once a user learns where to find an entry he can instantly get there without having to follow a trail leading him there as a normal menu or even a cascading radial menu would require (also referred to as mouse-ahead).

5.3.6. Implementation

The system runs on 5 networked PCs, one of which (the *head node*) accepts the tracking system output and distributes it over the 4 rendering nodes (2 per eye for 2 powerwall segments). Each node has access to the whole dataset so that only interaction/view changes need to be communicated, which is few enough data to be transmitted over normal gigabit ethernet without causing detectable lag. The existing hierarchical data structure consists of several recursive levels of spatial clusters, and the lowest level contains the actual particles. Each cluster also has an associated *representative* particle, which is rendered when no recursion into that cluster takes place. If the selection were an order of magnitude slower than the interaction itself, this would severely degrade the usability of the program and its acceptance by the users, so straight-forward traversal and flagging was not an option for the interaction extension. Thus the data structure was extended as follows: all clusters are tagged with the bounding box extents of all their children to enable correct hierarchical intersection and inclusion tests. To support fast selection and visibility operations that avoid a complete traversal of the hierarchy when possible, eight additional flags are stored in each cluster/point:

- t the element is temporarily selected
- t_c the element has temporarily selected children
- s the element is selected
- s_c the element has selected children

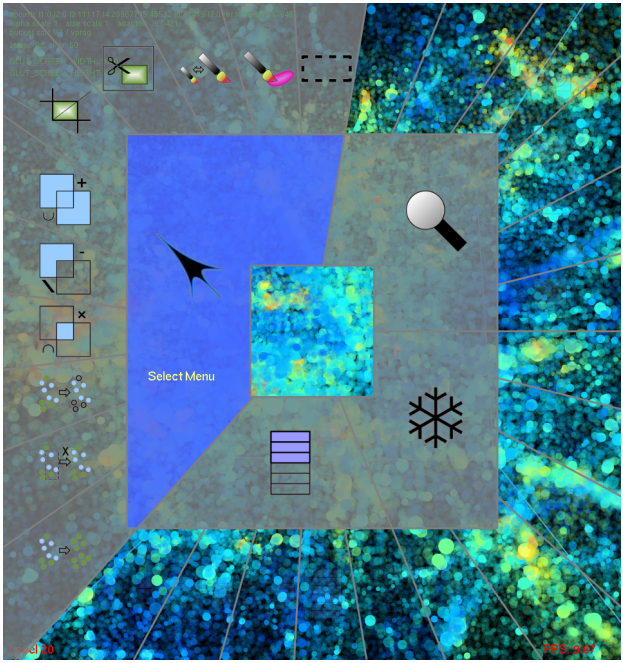


Figure 5.3.: The radial menu, blended over the dataset. Only submenu items in the selected menu are visible to reduce clutter.

- S_c all of the element's children are selected
- v the element is visible
- v_c the element has visible children
- V_c all of the element's children are visible.

To perform a temporary selection, the hierarchy needs to be traversed only where v_c holds. During the downstream recursion all points are tagged t if they are contained within the selection frustum. Upstream all clusters containing points $p|t$ are flagged t_c. To add or subtract the temporary selection from the permanent selection, only t_c are recursed, and the upstream is used for setting the flag s_c and removing t and t_c. For the intersection of both sets, both s_c and t_c need to be considered. To hide large numbers of points, the recursion can be stopped at the first cluster where the whole bounding box is inside the volume, since removing v, v_c, and V_c stops any further recursion and saves the effort of making all children consistently hidden (see also figure 5.4). To reinstate consistency upon showing the points again, only those parts have to

be traversed where V_c is not set.

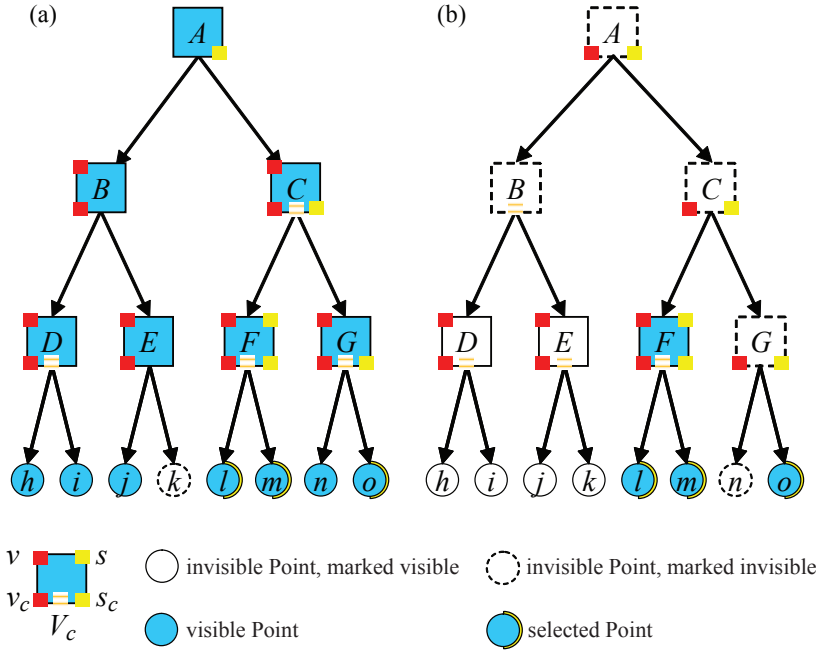


Figure 5.4.: (a) Example for a consistent hierarchy before triggering an interaction (b) Example for an inconsistent hierarchy when quickly hiding A completely. Only selected points remain. B is temporarily inconsistent because its children are only hidden because the recursion stops at B and the data below need not be modified.

One tricky part of the implementation was synchronizing the table interaction, because interaction basically has to be interpreted on the head node, but the user interface is displayed and used on the rendering nodes. The table was implemented with GTK [GTK], a free platform-independent toolkit. Although there are approaches which address multiple displays, like [dmx], it seemed possible that its limitation regarding OpenGL extensions could become a hindrance in the future and thus they were not used, instead relying on our own implementation. Basically the relative mouse position inside the GTK window is transferred from the master node to the clients and all mouse clicks are synthesized from the flystick interaction. As off-screen clicks – which are common since the table will usually span several rendering nodes' displays when shown on the powerwall – are often misinterpreted, the window is tem-

porarily moved to be completely visible (to the viewport origin) and the click triggered only then. The window's position is then reset to the earlier position after the click. The windows on all the screens would have to be identical as long as pixel positions are transferred. Using the mouse on the master node and showing a replica on a client this works well and is trivial. When it comes to using this approach while using the flystick, however, it will not work. The windows can still contain the same information, but the horizontal and vertical screen resolution cannot be guaranteed to be the same on the master as on the clients. As a result, fractional window-relative coordinates are transmitted from the master to the clients.

5.4. Discussion

In the following this user interface is analyzed from a theoretical point of view, its strength and weaknesses are pointed out. To streamline this analysis, the human action cycle is used as described for instance in [Nor88]. A number of exemplary interactions with the system are used to evaluate its properties. It is supposed to be clear by convention that the front button of the flystick is the 'main' button and thus semantically corresponds to a left-click on the mouse. Numbers in parentheses track the information the user has to keep in his short-term memory to effectively use the interface.

Inspecting a part of the dataset

Goal formation If the user wants to look at a particular point in the dataset, it requires the user to know which button to press to grab the dataset (1).

Execution stage To manipulate a dataset for inspection, the user just pulls the flystick closer, tilting it like he would the dataset, for example. This is a natural action as it would be executed on an object taken off a shelf in the real world, no additional information is required.

Evaluation stage Since the dataset is rendered at at least 10 FPS, the user has immediate feedback about his action and can also adjust his movements if the outcome does not match his intentions closely.

Singling out a data subset

Goal formation If the user wants to single out certain subsets of the data, he must first know which selection actions are possible. To get help, he can pop up the radial menu, which has icons that are assumed to convey the possible actions, mostly by convention from painting programs (brush, marquee). He must however remember which button pops up the menu (2).

Execution stage Selecting 'positively' and cropping the rest or selecting 'negatively' and hiding the selection makes no difference, the user can decide which he feels more comfortable with given the situation. The selection actuation button can be shown as a hint after choosing a menu item that toggles a selection mode, so it need not to be memorized. The user need not even have internalized the concept of a two-tiered selection for this purpose, as the temporary selection is enough to be cropped. Selecting with a brush or using a marquee can be accomplished as in other programs, the interaction should be obvious by convention.

Evaluation stage The immediate updating of the 3D scene shows the user whether his intention coincides with what has happened, and which additional adjustment of the dataset is required.

Repositioning the cursor

Goal formation The user notices that he cannot reach a part of the dataset. Either he decides to move so that he can reach it, or he repositions the mouse cursor, which however requires him to know that there is functionality to avoid moving himself.

Execution stage The user must remember the button to activate repositioning (3). Repositioning of the cursor and the user are natural movements and directly mapped to the 3D scene, so no special knowledge of the interaction is required.

Evaluation stage Feedback is immediate, so there is no gulf of evaluation.

Scrolling the attribute table

Goal formation If more points are selected than will fit on the screen, the user must scroll the attribute table to inspect their values.

Execution stage The user needs to know which button activates scrolling (4). After pressing it, the application can pop up a scrolling hint and the virtual horizon, so the user can see what will happen next when he tilts the flystick up or down. At the right edge of the window a jump icon is placed to remind the user that he can switch modes there.

Evaluation stage Scrolling happens immediately, thus providing feedback for the user.

Inspecting the attribute table

This basically only needs the user to know which button will toggle the table display, same as above (5).

Selection in the attribute table

Selection works in the same way as with a mouse interface and thus should be operable by convention.

Modifying the permanent selection

Goal formation The user wants to influence the selection that determines the content of the attribute table.

Execution stage It is only necessary to remember the button for the radial menu (same as for singling out items) and then follow the icons and their tooltips to perform operations that involve the temporary and the permanent selection.

Evaluation stage Since the temporary and the permanent selection have a different tint, they results can readily be observed. Modifying the permanent selection is, however, an unsafe operation as no memory is allocated to keep an undo buffer and thus marks an area where the tool could still be improved.

This demonstrates that most actions are designed to be intuitive. The effort of memorizing a total of 5 functions and their respective buttons should suffice for effectively using this interface, as the rest is either similar to a workstation interface or hinted at on the screen. This number is non-critical from the short-term-memory point of view [Mil56], however it means that the affordances of the application are not obvious. This could be corrected by overlaying the 3D scene with an iconic representation of the flystick (in a corner, for example), each of its buttons labeled with the action it will trigger (reposition cursor, toggle menu, select/trigger, toggle table). The mapping from real-world motion to what happens in the virtual scene is natural, as it mainly distorts distances but leaves directions the same, and is therefore easy to understand for the user. To verify these claims, an example scenario will be detailed below that serves as a context for some (informal) user tests aimed at proving the theoretically claimed suitability of the developed user interface.

5.5. Results and User Studies

Figure 5.5 shows a closeup of a cosmological simulation consisting of 20 million particles, where a freeform selection has been interactively drawn into the dataset. Figure 5.6 shows how the selection is perspectively extruded through the data as well as the holes caused by the spacing between selections. The adaptive rendering ensures about 10fps in both cases. Figure 5.1 finally shows a set of selected points and their data displayed in the table. Since the table and view are linked bidirectionally, the two selected points in the table are

highlighted by bounding boxes in the 3D view.

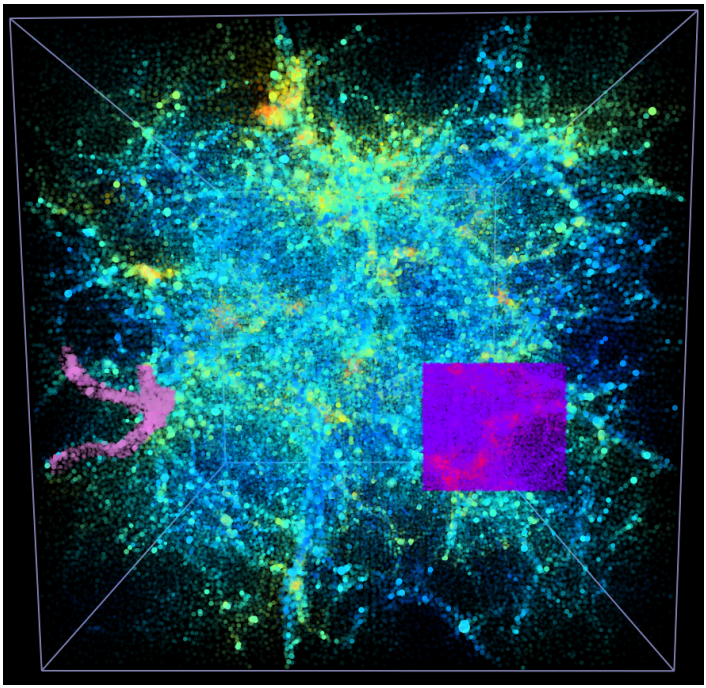


Figure 5.5.: Left: projected selection of a filament in the dataset, done with the brush tool. Right: projected rectangular selection. See also Fig. 5.6

Several informal user studies were conducted with few participants as suggested in [KHI+03], on the one hand to improve the problematic menu interaction in the VR environment and on the other hand to test different hypotheses, i.e.:

- *Usability depends on the visualized dataset.* Contrary to expectations, the complexity of the dataset has no negative impact on the usability. The users had to adjust the sensitivity of the positional mapping to better fit the amount of detail in the dataset, but they did not perceive that as a drawback.
- *Navigation using the flystick is more intuitive than using mouse and keyboard.* This was true, but only as long as the 3D cursor was inside the dataset and could easily be localized in space with stereo vision. Since the cursor is the interaction pivot, rotations around points outside the dataset put the users into difficulties at first. After some familiarization this effect

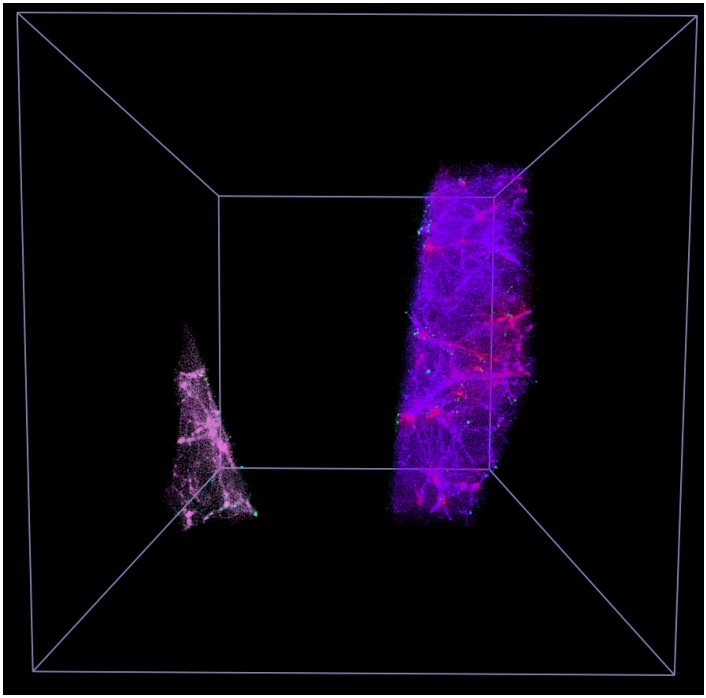


Figure 5.6.: Top-down view of the projected selections as in Fig. 5.5.

disappeared.

- *The rectangular selection is better suited for large areas.* Even worse, the users almost exclusively preferred this mode over the brush selection when working in single-display mode. In the VR environment, however, their behavior was inverted as most users preferred the combined candle-and-brush mode.
- *A two-tier selection is not intuitive.* Surprisingly, this feature was well accepted by the users, even though it is not supported by existing conventions and had to be explained at first.
- *Users with low to no experience using VR setups have more difficulty with the user interface than people that have already employed VR.* Since the flystick is quite different from space mice or pointing rods, there was nearly no difference in the learning curve and execution speed between experienced and inexperienced users.

In conclusion one of the targeted users was tested, i.e. a thermodynamics

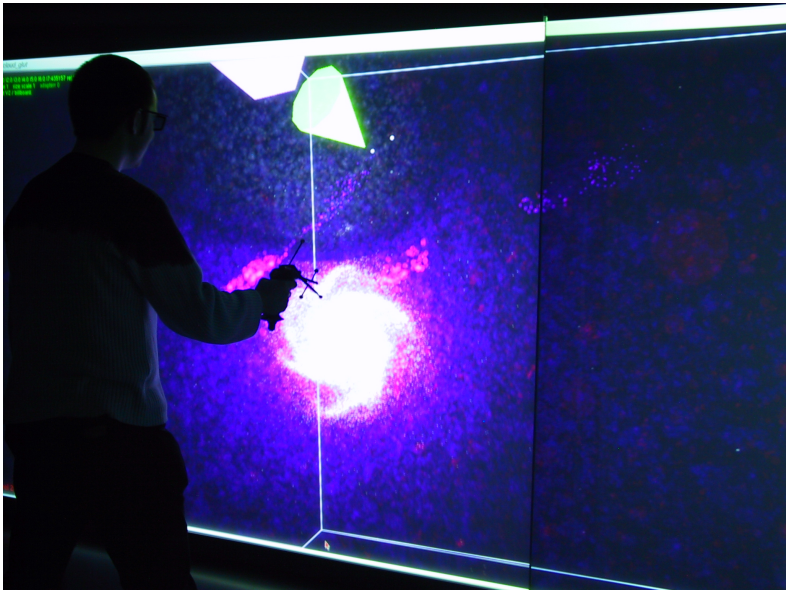


Figure 5.7.: One of the authors of the original paper navigating a galaxy simulation with the VR interface.

researcher with previous experience with VR environments. An aspect that was judged very positively was the navigation in the dataset by directly grabbing and rotating it using the flystick as a proxy for the interaction pivot, most probably due to the direct correspondence between user and dataset motion. The filtering possibilities also received positive feedback with a particular emphasis on the slicing aspect which allows the user to look inside clusters of molecules to grasp their density, and, in case of mixtures, the composition of agglomerates and the coating of surfaces. The user also pointed out that such an interface would be very suitable for a future simulation steering application.

6 Conclusion

The increasing power and flexibility of GPUs has long ago advanced its uses beyond the assistance in the rasterization stage as was common in the early days of PC graphics cards (referring to the first generations of 3dfx cards) to the more encompassing capabilities that support the CPU from the geometry stage onwards. Currently this trend is continuing far enough to allow several visualization approaches to work in a self-contained way on the GPU once the user has uploaded the source data. Examples include particle systems that can be simulated and visualized directly on the GPU, the same goes for flow simulation and visualization. This frees up the CPU, allowing for the streaming of out-of-core data, sorting, filtering, or the like, all in function of the main work accomplished on the GPU. Other compute-intensive approaches can also use the GPU as a coprocessor for numerical problems, which is even endorsed by the manufacturers themselves who nowadays provide specialized APIs that abstract from graphics and only provide the compute capabilities, but in a more streamlined fashion. Still the traditional way of using GPUs for the interactive visualization of data from various sources is probably the most widespread application area.

There are still limitations for what can be done on a GPU since for many problems data access and data generation cannot yet be implemented in a sufficiently flexible way. However, the algorithms that can be implemented making use of a GPU – if not exclusively – are becoming more and more numerous also as the cost of uploading and downloading data decreases, the parallelization of such code speeding up execution by more than the time consumed by the transferral operations. One such example was shown in section 4.7 where very simple, independent calculations were applied to a large dataset and the use of the GPU yielded a speedup of more than one order of magnitude.

Along the same line of thought, this work basically showed an efficient, GPU-centric implementation of the visualization pipeline with all its stages for the general problem of visualizing uncorrelated data, as defined in section 1.1.

6.1. Interaction and Filtering

The filtering introduced in chapter 5 offers basic set operations, however the interaction and workflow for it introduce novel concepts. Directly tracked three-dimensional input has been explored in related work with different input devices and the corresponding metaphors, however the 3D candle is new and very naturally mapped to the flystick. Its effects are also close to those of a real-world candle and as such easily picked up by the user. The developed two-tier selection is an efficient means for catching erroneous input in an immersive environment by providing a detailed preview of the effects of selection operations before actually altering the set. This concept can also be combined with undo support if the application can afford the necessary memory overhead. A strategy for working around the physiology-limited pointing precision has been proposed as well as an interface that combines iterative selection refinement and direct data inspection.

6.2. Mapping

Various algorithms for dimensionality reduction are available, however FastMap is still a reasonable choice when performance is an issue. A flexible system for user-steered dimensionality reduction relying on this algorithm and exploiting volume rendering for improved responsiveness has been introduced in chapter 4. A strong focus was set upon giving the user maximum flexibility to parameterize the algorithm. To speed up the dimensionality reduction even more, a GPU-assisted implementation of FastMap has been developed (in line with the general idea exposed above), making the whole mapping step truly interactive.

6.3. Rendering

The main focus of this dissertation is however the rendering of primitives with varying complexity from an implicit parameterized representation directly on the GPU. Basic and composite primitives were introduced in chapter 3 as well as a novel, complex, and costly-to-render tensor glyph. A thorough analysis of the parameters that affect performance has been given, also pointing out caveats for future implementations. Visualization strategies have been developed for the specific purpose of clustering analysis, for feature simplification in the 3D view that emphasizes the monomer-cluster interaction and a completely abstract view that shows inter-cluster interaction. This practical example can also prove that, and how, the proposed approach can be successfully applied to real problems.

6.4. Point-based Data Visualization Design

Consumer GPUs nowadays are so highly developed that the paradigm for using them is also shifting. In the very beginning, when 3Dfx started to popularize hardware that only offered support for the rendering stages from the rasterization onwards, the most important features were built around the texturing support that was mainly used for improving visuals in computer games while retaining decent performance. That particular capability, combined with blending, also led to GPU-accelerated volume rendering as soon as multitexturing and dependent lookups were supported. Over the years the concept of fetching data (from a texture), then doing a few simple operations with it before putting out the result was enhanced by ever more arithmetic operations in the 'slipstream' of the texture memory access, using the latency of the texture fetch operations to mask the cost of these operations. The flexibility and, even more important, the processing speed of GPUs is nowadays improved so much that a great number of computations can be performed exploiting those time slices even though the memory access is being accelerated in each new graphics processor as well. This achievable arithmetic intensity is becoming similar to that of CPUs except that the programs are still quite limited in size and the data access still is not very flexible. For example, the fragment shader is not capable of scattering (changing the destination position/address), while the geometry stage is severely restricted in gathering: the geometry shader can only access the direct neighborhood, the vertex shader no other vertices at all. Random reading from textures is possible for both. Given the concept that programs have a certain cost in either time *or* space, the trend should obviously go in the direction of replacing data by a parameterized shape that can be expanded on the fly making use of the high processing power, on the one hand because graphics card memory is more costly and thus tends to be limited more often than the main memory, and on the other hand to spend less time transporting the data from the CPU to the GPU, which is a very important factor when dealing with time-dependent data, for instance. Implicit geometry is thus a very important concept that allows to effectively exploit the resources of modern programmable GPUs. At the time of development, the tubelet

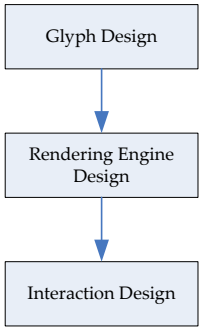


Figure 6.1.: Basic workflow for designing a specific implementation of the visualization pipeline.

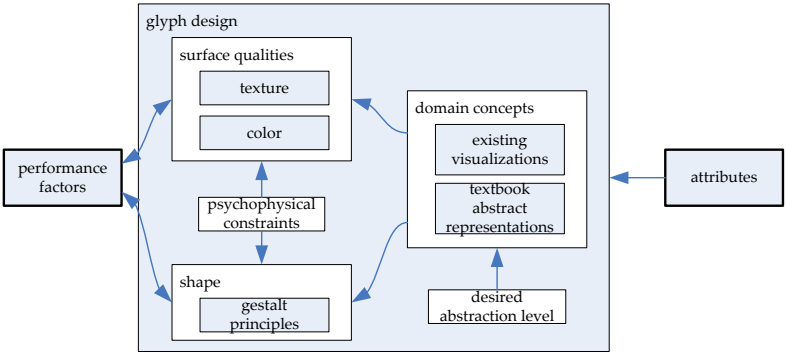


Figure 6.2.: Glyph design parameters and influential factors.

shader (see section 3.3) was still very much on the brink of being far to expensive to use with anything but the most powerful graphics cards, but nowadays even full-screen output of large datasets can be generated at interactive frame rates. This concept is contrary to what has been common practice for several years now, that is, mostly pre-computing as much as possible and thus replacing processing time by memory requirements to obtain an interactively manipulable visualization. As a consequence, a visualization application design process for point-based data is proposed that takes into account several factors that influence mostly the resulting performance. It summarizes the approach refined over time and used throughout this dissertation.

To obtain an efficient and above all effective implementation of the visualization pipeline for a specific research area and the respective domain experts as users, the following workflow is proposed (see figure 6.1). First of all, a glyph, a representative for a single data point, is needed, and this must mainly be derived from the application context. In function of this decision, the whole rendering engine must be designed, defining how the geometry will be generated and how the workload is to be split between CPU and GPU. Finally, the interface to the user should be worked on, ensuring the parametric control over the different pipeline stages in an intuitive way.

6.4.1. Glyph Design

The glyph design is driven by the largest number of parameters in the whole development process. A basic concept is needed first, which is most conveniently derived from previous work especially in the application domain, as to be easier to understand and more intuitive (if only by convention) to the end users. Such inspiration can either be taken from existing computer-based visu-

alization approaches that might still exhibit flaws which can be especially improved upon, or from legacy visualizations such as illustrative works in textbooks, which usually focus on details and might at first glance not be suited for the complexity of the task at hand. Recognition is very powerful though and will increase the acceptance from domain users significantly, so it would be worth the time employed to simplify and streamline overly detailed glyphs, for example (this is represented by the 'desired abstraction level' influence in figure 6.2). This very strategy was employed for the molecular visualization in chapter 3, which improved mainly the implementation and offered a similar, but higher-quality output than an already existing approach, based on concepts that are known from textbooks. The tensor glyphs in that chapter are, on the other hand, a more abstract representation of the ellipsoid-based visualization of tensors (which is often seen in related work) and could be seen as the interpolated hull of densely packed ellipsoids. Whether the glyph is similar to existing representations, a novel (abstract) one or even the real-world counterpart of a datapoint also depends on the number of attributes that need to mapped onto it. Even with a real-world counterpart, some attributes (like speed or energy/temperature) do not have a directly visible manifestation so to make them all visually distinguishable, either the basic shape needs to be altered to accommodate more features or an abstract representation needs to be developed. Features can be roughly classified into shape (the geometry itself) and surface qualities (see figure 6.2). To make sure that the final result can work, the psychophysical constraints of the human beholder must be taken into account, for example the gestalt principles [Wer23] to make sure that glyph instances can be distinguished from each other (inter-glyph), if so desired, as well as the different features of a glyph can be distinguished (intra-glyph), resulting in an upper limit of features. If this limit is hit, the directly visible attribute set must be restricted. Symmetries and ambiguities must be taken care of as well, as for dipoles the different van-der-Waals radii and the charge-induced coloring make its orientation unambiguous.

The attributes themselves must be addressed specifically (figure 6.3). In many occasions their sheer number is very hard to fit onto a glyph or suffers from side-effects. The first problem can be addressed by interactively subsetting the available attributes via user interaction (the 'filtering' pipeline stage) or by working on automatically generated subset proposals, as long as the process can be steered. A simple example for the second effect might include an attribute that represents a size (and thus is mapped onto it to ease recognition by convention) and another attribute whose value is the more interesting the smaller the size gets. This classical quandary requires measures to be taken to still offer a user the possibility of quickly inspecting the second attribute after detecting the candidate data points in an overview visualization. A number of possibilities can then be explored: coordinated views, some sort of assisted singling-out interaction, like a fish-eye zoom [Fur86], or di-

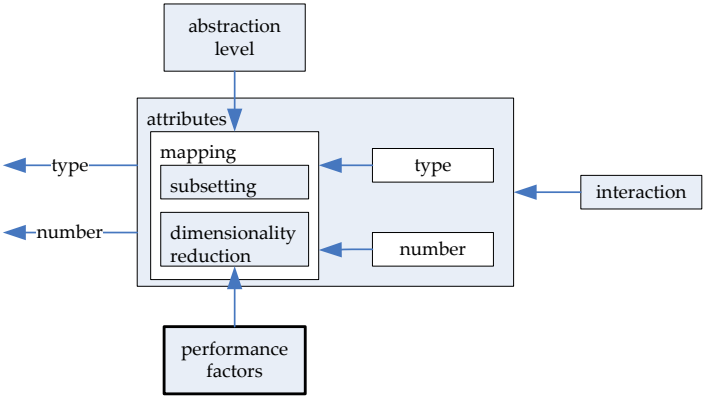


Figure 6.3.: Scheme for supplying attributes to the visualization pipeline. Whether the input is directly passed or filtered or mapped to a subset or abstract representation is a user decision.

rect zoom [RLLEE03] on key-press, mouse-over, or similar. An orthogonal approach would trade the intuitive mapping for another one that results in the required 'contrast' for the task at hand (keeping the size fixed and highlighting small items by color, for example). The third option is to reduce the available attributes to an easily displayable number, or just a position which consequently groups the available data in an abstract similarity space (see also chapter 4). Glyph features can still be used to display the interesting attributes, but all small items are ideally grouped, so after finding just one, its neighbors can be conveniently inspected (and also more easily compared due to spatial proximity) afterwards. If it is not possible to leave that choice to the user for some reason, a user study should be conducted to find out which alternative provides the best solution for the problem at hand.

Another external glyph design factor is the performance after implementation, which might lead to a need for simplification to ensure interactive performance, for example, or level-of-detail methods which ease the distinguishing of glyphs at higher distances and still make the full range of attributes readable when the glyph is zoomed in. This will be discussed in more detail below.

6.4.2. Rendering Engine Design

The rendering engine design decisions are heavily interdependent on several performance factors, which in turn are not independent themselves (see figure 6.4 and figure 6.5). Many of these effects have been shown in chapter 3 along with concrete performance figures that demonstrate these pitfalls in a

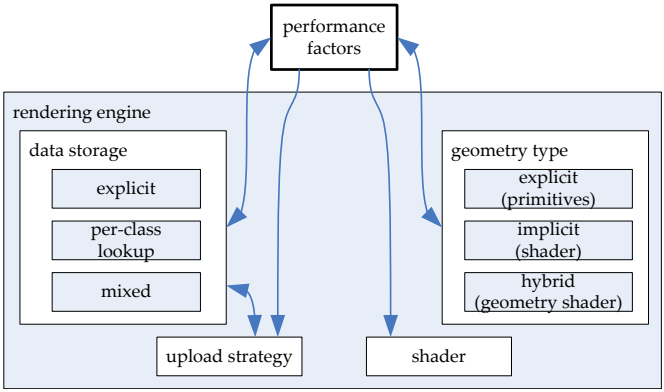


Figure 6.4.: Different options for the overall rendering engine design, including the external performance factors.

fully functional visualization system. The first, fundamental, choice to make is the representation of the glyph geometry. The traditional tesselated-geometry approach (explicit) has the highest impact on available memory, its bandwidth and above all upload cost for the transfer onto the GPU. Additionally the surface and silhouette quality are inferior to the other approaches. The completely implicit representation instead relies heavily on the arithmetic power of the GPU, but can offer the highest quality (as long as the numerical precision of the GPU is sufficient), while hybrid approaches try to compensate especially costly shading with additional geometry or by providing an already better approximation of the geometry, which can help in certain cases (see section 3.2.2) but is highly situational. The GPU-based raycasting trades computation time for a maximally concise parametrization of each glyph, instead of space for time, as mentioned before (see also [Wei05]). Such a hybrid object space/screen space approach, on the other hand, depends most heavily not only on the data size, but also on the output size (see section 3.2.2). This paradigm shift was only possible because of the steadily increasing processing power of GPUs. Its suitability for a number of glyphs of varying complexity has been shown in chapter 3, however the decision must obviously be made anew for each glyph depending on the suitability of its surface for analytic or iterative raycasting. For very costly iterative approaches a combination with the previous pre-computed solutions could also provide lookup data to cut down on iterations. So the more complex the glyphs become, the more options need to be explored and balanced out for optimal performance.

The data storage affects and is affected by several performance factors. It seriously reduces the available memory if no categorization is possible and all

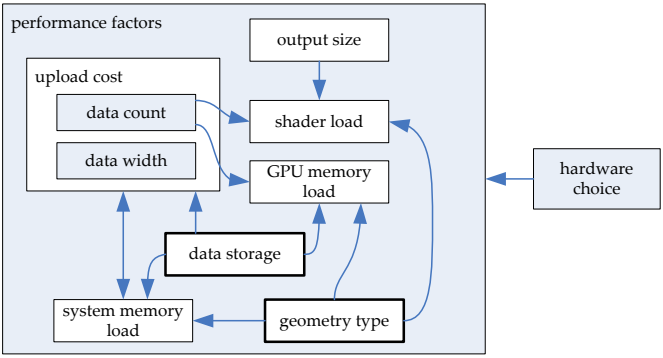


Figure 6.5.: Performance factors affecting the visualization.

values have to be stored explicitly per point. This in return also adversely affects flexibility if vertex buffer objects are to be used, but variable subsets of the attributes are used at any one time (see section 3.2.2). On the other hand, only the most current graphics hardware can significantly benefit from parts of the attributes being stored in textures, so an optimal all-round solution is very difficult to obtain. The most conspicuous problem that surfaced, however, remains the fact that no really satisfactory VBO performance could be obtained, regardless of mode and glyph, at least when comparing it with vertex arrays. This bodes especially ill for the next version of OpenGL, if the vertex array interface will actually be removed.

The hardware on which the system will run also affects several performance factors. Obviously the newer the hardware is, the better the performance gets, with the restriction that basically only Nvidia cards work reliably with the OpenGL API, as ATI drivers have notoriously problematic drivers and performance issues especially with point-based algorithms (at least when it was tested every now and then during this work). The CPU/chipset combination however can also have interesting effects, since AMD64 is the only platform that offers enough bandwidth to allow the usage of quad primitives instead of points with no negative performance impact (see section 3.2.2).

6.4.3. Interaction Design

Figure 6.6 summarizes the aspects that need consideration when designing the interaction for the developed application. The available hardware is mainly a cost factor – and it was as well for this work – that limits the choice of the physical interface, but this does not pose a problem in itself as long as a fitting metaphor is developed that allows users to easily understand and use the in-

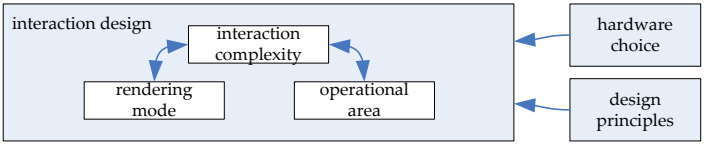


Figure 6.6.: Design factors for the user interaction component of the visualization.

terface (see chapter 5). The rendering mode refers to the option of using stereo-enabled output, which mainly improves the depth perception for the dataset and the 'cursor' and as such gives manipulation in 3D enough feedback to be usable. For conventional output devices interaction on the near clipping plane projected into depth is still a much safer approach. The interaction complexity itself also accounts for a good part of the design effort that is required. Simple actions, like orientation changes for a dataset, should require as few button presses and mental mapping steps as possible, the more direct an interaction is, the better. However complex input requires more program-side support that saves the user time while not hindering him in pursuing his intentions.

The last aspect to consider is the operational area, that is whether the application is to be used on conventional workstations which usually offer the common keyboard-and-mouse interface, or if it will mainly run in VR installations. It might thus be helpful to accept certain restrictions as to the input hardware and balance that out with a careful design of the user interface and certain interaction aids (for example the selection of overlapping items as in [RLLEE03]) instead of designing for the rare case in which a user actually has special hardware at his disposal and penalize all users who only have access to workstations. In any case the principles established in the HCI community must always be taken into account, following the rules of Shneiderman, the concepts of Norman, or even the EN ISO 9241 Part 10, which are still the most important basis for user design decisions.

6.5. Outlook

The work presented in this dissertation mainly relies on the fact that GPUs offer parallelization and specifically tailored arithmetic capabilities to a degree that the more generic CPUs and especially their more flexible overall system design will not be able to offer for quite some time. Since the technological advances in this area show no tendency of significantly slowing down, the general direction pointed out here can be followed for a while even in the future. This means that it will be increasingly realistic to generate even more complex geometry on the fly from an implicit representation, even with costly iterative

approaches. A challenge that will be most probably solved next are protein surfaces, for example. These represent the accessibility by a certain solvent in which the protein is immersed and are calculated by 'rolling' a probe molecule over the protein. The resulting primitives are multiple spherical triangles and sections of the inner part of a torus. The spherical triangle glyph basically requires an efficient-to-compute boundary parametrization and a tightly fitting bounding geometry (a tetrahedron comes to mind). Research on the torus has failed in the visualization community up to now. No satisfying result could be obtained and thus the approach has not been published yet: the implicit approach exceeded the maximum complexity of fragment shaders at the time, and employing the geometry shader to tessellate the geometry did not improve on the CPU implementation since current hardware is lacking optimization in the geometry shader stage.

Another area where improvement will probably be seen is medium-scale parallelism. Clusters with GPUs for multiple levels of parallelism are very common for volume rendering or other algorithms which benefit significantly from image-space parallelism (like the hybrid approaches shown in this thesis). However, the very cost-effective approach of multiple GPUs in a single machine ('SLI mode' as it is named for Nvidia cards) still is lacking complete support from the driver side at the time of writing. Thusly equipped systems do work without problems, however there are still performance issues. One glaring example is the nonexisting driver-side parallelism when texture upload is concerned. However, such problems will surely be solved in time so that tandems of a CPU core and a GPU can be formed on such machines to optimally make use of the available hardware and also allow for finer adjustment of the cost/performance factor: when image-space parallelism is concerned, the probability is very high that two average cards will outperform a single high-end card, but cost the same or even less. This obviously requires a higher implementation effort, but will also help with obtaining the maximum possible performance by just using several high-end cards.

A Acronyms

AGP	Accelerated Graphics Port
API	Application Programming Interface
AR	Augmented Reality
ARB	Architecture Review Board
CAVE	CAVE Automatic Virtual Environment
CFD	Computational Fluid Dynamics
CG	C for Graphics
CPU	Central Processing Unit
CT	Computed Tomography
CUDA	Compute Unified Device Architecture
DFG	Deutsche Forschungsgemeinschaft, the German Research Foundation
DIN	Deutsches Institut für Normung, the German Institute for Standardization
DTI	Diffusion Tensor Imaging
DVR	Direct Volume Rendering
FA	Fractional Anisotropy
FPS	Frames per Second
FSB	Front-Side Bus
GL	short for OpenGL
GLSL	GL Shading Language
GPGPU	General-purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
GTK	The GIMP Toolkit
HCI	Human-Computer Interaction
HD	High Definition (i.e. 1920x1080 resolution)
HLRS	Höchstleistungsrechenzentrum Stuttgart, the High Performance Computing Center Stuttgart
HLSL	High Level Shading Language
ID	Identifier

IEEE	Institute of Electrical and Electronics Engineers
IR	Infrared
ISO	International Organization for Standardization
ITB	Institute of Technical Biochemistry
ITT	Institute of Thermodynamics and Thermal Process Engineering
LIC	Line-Integral Convolution
LOD	Level Of Detail
MB	MegaByte, intended as 2^{20} bytes
MD	Molecular Dynamics
MDS	Multi-Dimensional Scaling
MRI	Magnetic Resonance Imaging
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PDA	Personal Digital Assistant
RAID	Redundant Array of Inexpensive Drives
RAM	Random-Access Memory
RBF	Radial Basis Function
RGB	Red, Green, Blue
RGBA	Red, Green, Blue, Alpha
ROI	Region of Interest
SATA	Serial Advanced Technology Attachment
SFB	Sonderforschungsbereich (collaborative research centre)
SIGGRAPH	Special Interest Group on Graphics
SIMD	Single Instruction, Multiple Data
SLERP	Spherical Linear Interpolation
SLI	Scalable Link Interface
SPH	Smoothed Particle Hydrodynamics
SPP	Schwerpunktprogramm
VBO	Vertex Buffer Object
VPU	Vector Processing Unit
VR	Virtual Reality

B Additional Performance Figures

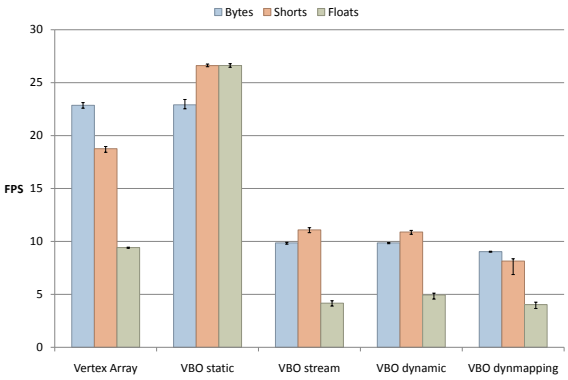


Figure B.1.: Upload performance for 4 million points on the Pentium 4 machine equipped with a GeForce 6800.

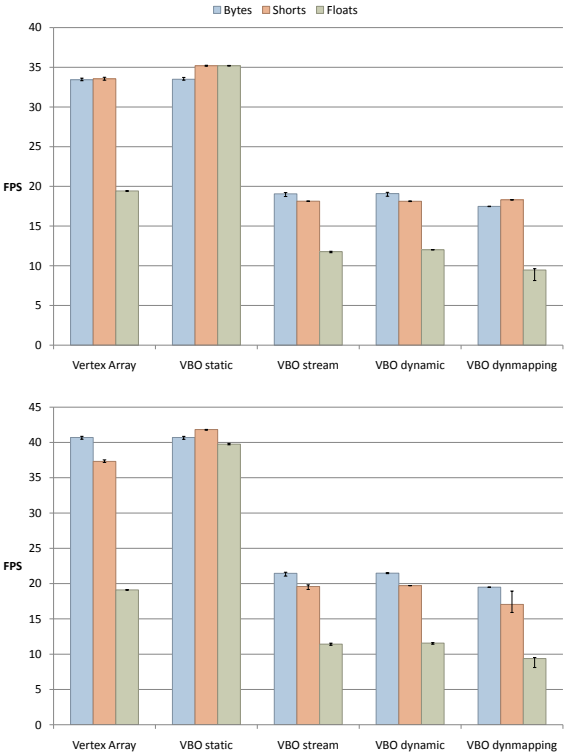


Figure B.2.: Upload performance for 4 million points on the AthlonXP machine. Top: GeForce 6800GS, bottom: 7900GT.

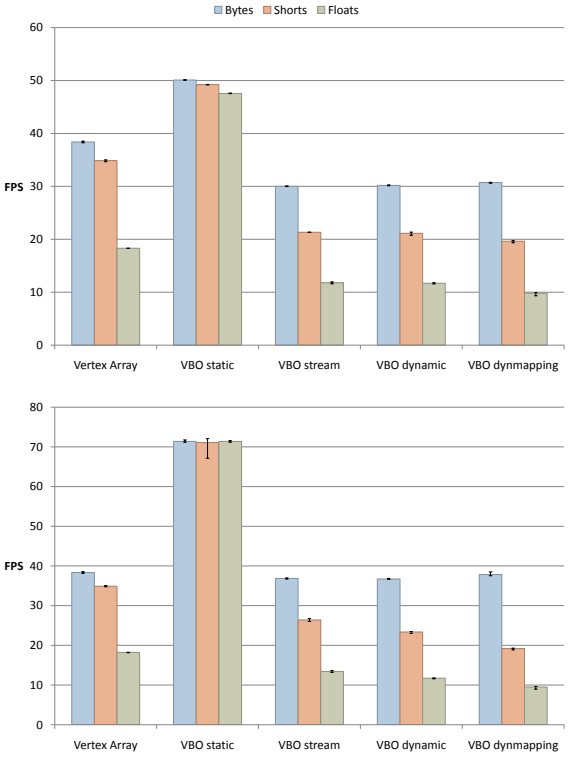


Figure B.3.: Upload performance for 4 million points on the AthlonXP machine. Top: GeForce 8600GT, bottom: 8800GTX.

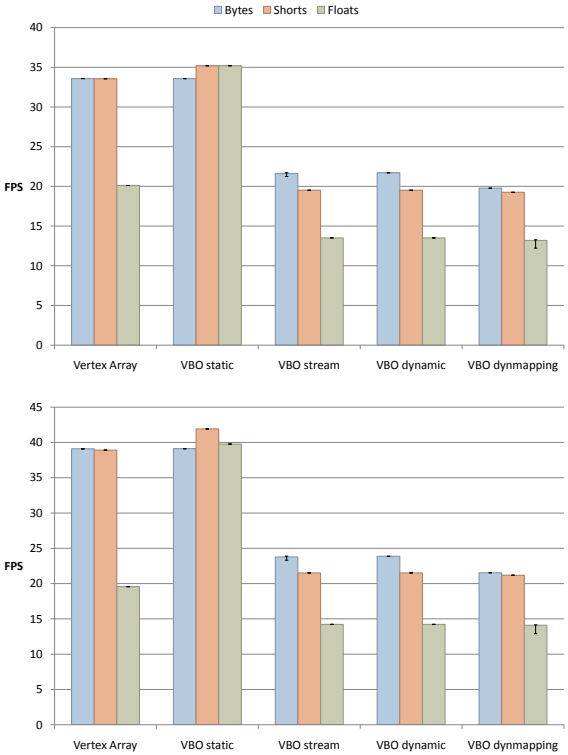


Figure B.4.: Upload performance for 4 million points on the Phenom machine. Top: GeForce 6800GS, bottom: 7900GT.

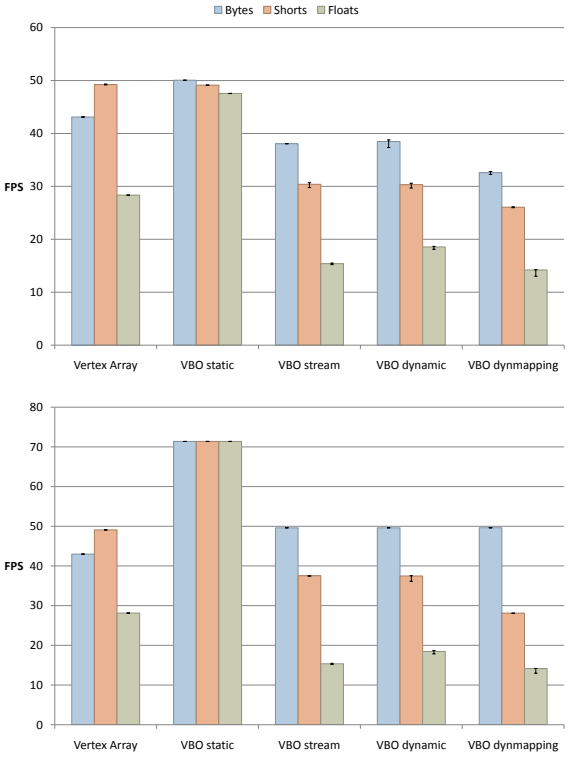


Figure B.5.: Upload performance for 4 million points on the Phenom machine. Top: GeForce 8600GT, bottom: 8800GTX.

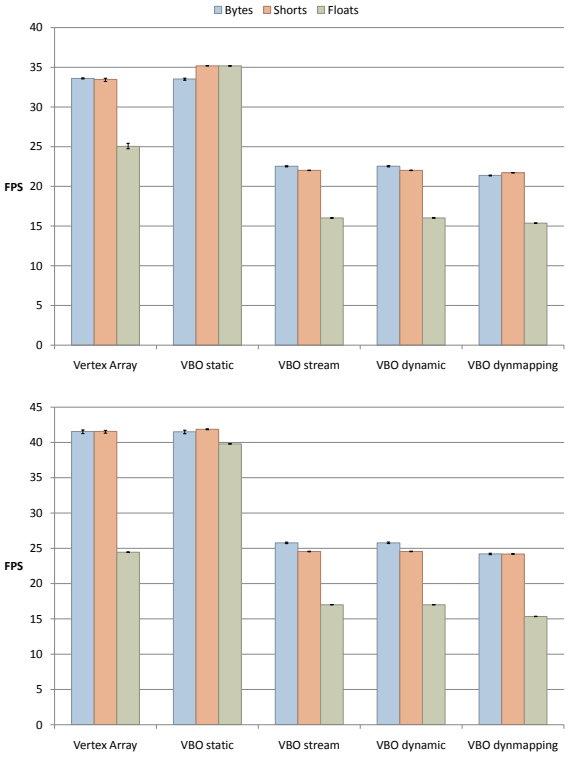


Figure B.6.: Upload performance for 4 million points on the Core2 Duo machine. Top: GeForce 6800GS, bottom: 7900GT.

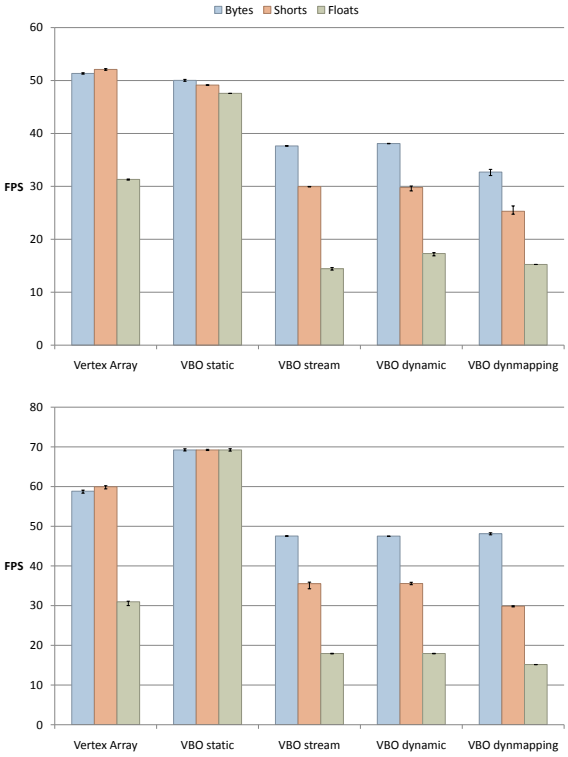


Figure B.7.: Upload performance for 4 million points on the Core2 Duo machine. Top: GeForce 8600GT, bottom: 8800GTX.

C German Abstract and Chapter Summaries

Eines der verbreitetsten Anwendungsgebiete für computergestützte Visualisierung ist die Wissenschaft. Forscher müssen mit Datensätzen unterschiedlicher Art arbeiten, die jedoch alle die gemeinsame Eigenschaft besitzen, dass sie in Rohform die kognitive Kapazität eines Menschen bei weitem übersteigen. Die Visualisierung versucht daher nicht nur, Benutzer zu befähigen, so schnell wie möglich sehr viel Informationen aus solchen Datensätzen zu extrahieren, sondern es dem Forscher überhaupt zu ermöglichen, mit Datenmengen umzugehen, die zu groß und komplex für die direkte Erfassung mit der menschlichen Sensorik sind. Diese Arbeit konzentriert sich auf unkorrelierte Punktdaten, oder auch Punktwolken, die in experimentell erfasst oder aus Computersimulationen gewonnen sind. Solche Datensätze sind gitterlos und enthalten keine Konnektivitätsinformationen, und jeder Punkt stellt eine Entität für sich dar. Um effektiv mit solchen Datensätzen arbeiten zu können, müssen hauptsächlich zwei Probleme gelöst werden: einerseits muss eine große Zahl komplexer Primitive mit einer potentiell hohen Anzahl von Attributen visualisiert werden, und andererseits muss die Interaktion mit diesen Objekten einfach und intuitiv sein.

In dieser Dissertation werden neue Methoden präsentiert, die große, punktbasierte Datensätze mit hoher Dimensionalität handhabbar machen. Der Beitrag zum Rendering hunderttausender applikationsspezifischer Glyphen ist eine GPU-basierte Lösung, die die Exploration von Datensätzen mit einer moderaten Anzahl von Dimensionen, aber einer extremen Anzahl von Punkten, ermöglicht. Es wird gezeigt, dass dieser Ansatz sowohl für Molekulardynamikdatensätze, als auch für 3D-Tensorfelder sinnvoll ist. Performance-kritische Einflussfaktoren dieser Algorithmen werden ausführlich analysiert, mit dem Hauptaugenmerk auf dem schnellen Rendering der komplexen Glyphen in hoher Qualität. Um die Visualisierung von Datensätzen mit vielen Attributen und einer geringeren Anzahl von Datenpunkten zu verbessern, werden Methoden zur interaktiven Dimensionsreduktion und zur Analyse der Einflüsse unterschiedlicher Dimensionen und Metriken vorgestellt. Die Darstellung der resultierenden Daten in einem 3D-Ähnlichkeitsraum wird ebenfalls behandelt. Die interaktive Parametrierung der Reduktion wurde durch

eine GPU-basierte Implementierung erreicht, die es erlaubt, die Auswirkung in Echtzeit zu begutachten.

Mit der Verfügbarkeit einer schnellen Visualisierung fehlt als letzte Komponente zu einer durchgängigen Implementierung der Visualisierungspipeline die Mensch-Maschine-Interaktion. Der Nutzer muss in die Lage versetzt werden, im Informationsraum zu navigieren und mit dem Datensatz zu interagieren, indem er interessante Daten selektiert oder durch Filterung hervorhebt, und die Attribute spezieller Datenpunkte einsieht. Heutzutage muss der Anwendungskontext und die Modalität der unterschiedlichen Interaktionskonzepte unterschieden werden. Die Forschung geht hier von Tastatur-und-Maus-Interaktion auf dem Desktop über verschiedene haptische Interfaces (die mitunter eine Rückkopplung erlauben) bis zur Tracking-basierten Interaktion bei VR-Installationen.

Im Kontext dieser Arbeit wird das Interaktionsproblem für zwei unterschiedliche Szenarien angegangen: das Erste ist die Workstation-basierte Analyse des Clustering-Verhaltens in Simulationen aus der Thermodynamik, das Zweite eine immersive Virtual-Reality-Navigation und Interaktion mit Punktwolkendatensätzen.

Kapitelübersicht

Kapitel 1

Dieses Kapitel beschreibt den Kontext dieser Dissertation, deren Hauptfokus auf großen, Punkt-basierten Daten mit mehreren Attributen pro Punkt liegt. Es wird begründet, warum speziell drei Hauptaspekte (GPU-Nutzung, Dimensionsreduktion und Interaktion) behandelt werden. Die Projekte und Kooperationen, auf Basis derer die Arbeit enstanden ist, werden ebenfalls erläutert.

Kapitel 2

Die Visualisierungspipeline und ihre unterschiedlichen Abschnitte werden erklärt, da diese Struktur auch den roten Faden dieser Arbeit im Gesamten ausmacht. Die unterschiedlichen Datentypen, die die Grundlage für die unterschiedlichen Visualisierungsalgorithmen darstellen, werden vorgestellt und mit den folgenden Kapiteln in Zusammenhang gebracht. Die Funktionalität und Programmierbarkeit moderner Graphikhardware wird erklärt, und die Grundlagen des Volumenrendering ebenso wie der Hintergrund der Mensch-Maschine-Kommunikation kurz erläutert.

Kapitel 3

In diesem Kapitel wird die effiziente Darstellung großer Datensätze mit Hilfe von GPU-generierten Glyphen behandelt. Diese Glyphen basieren auf impliziten geometrischen Oberflächen. Die verwendeten Datensätze aus Molekulardynamiksimulationen werden zusammen mit den gebräuchlichen Visualisierungsansätzen und anderen themenbezogenen Arbeiten beschrieben. Zunächst wird eine gründliche Analyse der verfügbaren Möglichkeiten für die Darstellung und den Transport der Daten zur Graphikkarte durchgeführt. Das optimierte Rendering wird erläutert und mehrere einfache und zusammengesetzte Glyphen vorgestellt. Die daraus entwickelte Applikation für die zeitbasierte Visualisierung von MD-Datensätzen ist das Ergebnis dieser Vorarbeiten, die zusätzlich durch eine abstrakte Visualisierung die Erforschung der Nukleation von Gasen ermöglichen soll, in diesem speziellen Falle die Detektion von Tröpfchen und die Verfolgung ihrer Entwicklung über die Zeit. Als weiteres Anwendungsfeld wird auf die Diffusionstensorvisualisierung im Kontext der Medizin eingegangen. Ein sehr komplexer Glyph zur Repräsentation des gesamten Eigensystems eines solchen Tensors und dessen effiziente Implementierung werden vorgeschlagen. Ergebnisse mit Datensätzen aus der Medizin belegen die Angemessenheit dieses Ansatzes.

Kapitel 4

Da manche Datensätze eine zu hohe Zahl von Attributen ausweisen, um alle gleichzeitig und verständlich darstellen zu können, wird für solche Fälle die interaktive Reduktion der Dimensionen auf ein Ähnlichkeitsmaß vorgeschlagen, um dann die allgemeine Datensatzstruktur auf Gruppierungen und Trends untersuchen zu können. Die Darstellung der reduzierten Daten ist mittels Volumenrendering realisiert, um eine konstante minimale Performance unabhängig von der Datensatzgröße garantieren zu können. Beispielhaft wird diese Vorgehensweise auf einen öffentlich verfügbaren hochdimensionalen Datensatz angewendet, auch um das Potential visueller Abfragen anhand unterschiedlicher Transferfunktionen in der Volumendarstellung aufzuzeigen. Dieser Ansatz basiert auf der iterativen Verfeinerung und Optimierung der untersuchten Daten durch den Benutzer und erfordert daher höchste Performance bei der Dimensionsreduktion. Dies führte zu einer GPU-beschleunigten Implementierung des verwendeten Algorithmus (FastMap). In diesen Abschnitt werden auch einige Details der Implikationen des Texturmanagements erläutert, die während der Optimierung des Programms auffielen.

Kapitel 5

Dieses Kapitel befasst sich mit dem Filterungsschritt der Visualisierungspipeline, im speziellen der interaktiven Filterung durch unterschiedliche Selektionsmechanismen sowohl auf normalen Workstations als auch auf VR-Displays mit Tracking-basierter Interaktion. Im Besonderen werden die Implikationen letzterer behandelt, zum Beispiel die stärkere Ermüdung, die Eingabegenauigkeit, und auch Benutzungsfreundlichkeit bei komplexen Interaktionen. Eine neuartige interaktive Filter-Metapher zur Darstellung von mehr Details, die auch zur Verbesserung der Renderingperformance verwendet werden kann, wird vorgestellt. Ihre Benutzbarkeit wird unter Gesichtspunkten der Prinzipien nach Norman und auch durch eine informelle Benutzerstudie geprüft und als effektiv und effizient gezeigt.

Kapitel 6

Das letzte Kapitel fasst die Beiträge dieser Dissertation zusammen und erklärt den Prozess, auf Grund dessen die hier vorgestellte Implementierung der Visualisierungspipeline erarbeitet wurde, aus der Sicht eines Softwaretechnikers. Die Einflussfaktoren, die berücksichtigt werden müssen, wenn eine solche Lösung entwickelt wird, werden mit ihren Verknüpfungen untereinander aufgezeigt und die Aussagen anhand der verschiedenen Teile dieser Arbeit erläutert. Ein kurzer Ausblick auf mögliche zukünftige Entwicklungen wird ebenfalls gegeben.

List of Figures

List of Tables

Bibliography

[acc] Discovery Studio. http://www.accelrys.com. 37

[AF98] Z. Ai and T. Fröhlich. Molecular dynamics simulation in virtual environments. *Computer Graphics Forum*, 17:267–273, 1998. 39

[AHK01] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional space. *Lecture Notes in Computer Science*, 1973:420, 2001. 104

[AKK96] M. Ankerst, D. A. Keim, and H.-P. Kriegel. Circle segments: A technique for visually exploring large multidimensional data sets. In *Proceedings of IEEE Visualization '96*, 1996. 96

[ami] amira. http://www.amiravis.com/. 37

[And72] D. F. Andrews. Plots of high-dimensional data. *Biometrics*, 28(1):125–136, 1972. 23

[AT87] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, 1987. 35

[AVS] AVS. http://www.avs.com. 37

[BBMP97] M. Billinghurst, S. Baldis, L. Matheson, and M. Philips. 3d palette: a virtual reality content creation tool. In *VRST '97: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 155–156, 1997. 123

[BC87] R. Becker and W. S. Cleveland. Brushing scatterplots. *Technometrics*, 29(2):127–142, 1987. 96

[BDST04] C. Bajaj, P. Djeu, V. Siddavanahalli, and A. Thane. Texmol: Interactive visual exploration of large flexible multi-component molecular complexes. In *Proceedings of the conference on Visualization '04*, pages 243–250, 2004. 34, 38, 39, 161

[Bea02] C. Beaulieu. The basis of anisotropic water diffusion in the nervous system - a technical review. *Nuclear Magnetic Resonance in Biomedicine*, 15:435Ű–455, 2002. 79

[Bec97] B. G. Becker. Volume rendering for relational data. In *Proceedings Information Visualization '97*, pages 87–90, 1997. 99

[Bel61] R. E. Bellman. *Adaptive Control Processes*. Princeton Univ. Press, 1961. 95

[BETT98] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, 1998. 72

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH 2004*, 2004. 28, 119

[BG97] I. Borg and P. Groenen. *Modern Multidimensional Scaling*. Springer Verlag, New York, 1997. 18, 96

[BG98] D. Brodbeck and L. Girardin. Combining topological clustering and multidimensional scaling for visualising large data sets. accepted for, but not published in Proceedings of IEEE Information Visualization 1998, 1998. 97

[BHR+94] M. Brill, H. Hagen, H.-C. Rodrian, W. Djatschin, and S. V. Klimenko. Streamball techniques for flow visualization. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 225–231, 1994. 66

[BHZK05] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. In *Proceedings of Point-Based Graphics '05*, pages 17–141, 2005. 34

[BK03] M. Botsch and L. Kobbelt. High-quality point-based rendering on modern GPUs. In *Pacific Graphics'03*, pages 335–343, 2003. 34

[Bly06] D. Blythe. The direct3d 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, 2006. 24, 27

[BML94] P. J. Basser, J. Mattiello, and D. LeBihan. Estimation of the effective self-diffusion tensor from the nmr spin echo. *Journal of Magnetic Resonance, Series B*, 103(3):247–254, 1994. 90

[BP96] P. J. Basser and C. Pierpaoli. Microstructural and physiological features of tissues elucidated by quantitative-diffusion-tensor mri. *Journal of magnetic resonance. Series B*, 111(3):209–219, 1996. 81

[BPP+00] P. J. Basser, S. Pajevic, C. Pierpaoli, J. Duda, and A. Aldroubi. In vivo fiber tractography using dt-mri data. *Magnetic Resonance in Medicine*, 44:625–632, 2000. 80

[BRB+07] K. Bidmon, G. Reina, F. Bös, J. Pleiss, and T. Ertl. Time-Based Haptic Analysis of Protein Dynamics. In *Proceedings of World Haptics Conference (WHC 2007)*, pages 537–542, 2007. 3, 20

[BSK04] M. Botsch, M. Spernat, and L. Kobbelt. Phong splatting. In *Proceedings of Symposium on Point-Based Graphics 2004*, pages 25–32, 2004. 34

[BSW06] P. Baudisch, M. Sinclair, and A. Wilson. Soap: a pointing device

that works in mid-air. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 43–46, 2006. 19

[BSW07] P. Baudisch, M. Sinclair, and A. Wilson. Soap: how to make a mouse work in mid-air. In *CHI '07: CHI '07 extended abstracts on Human factors in computing systems*, pages 1935–1940, 2007. 30

[Buc03] I. Buck. Data parallel computing on graphics hardware. In *Graphics Hardware '03*, 2003. 119

[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*, pages 53–64, 2002. 34

[CC92] M. Chalmers and P. Chitson. Bead: Explorations in information visualization. In *Research and Development in Information Retrieval*, pages 330–337, 1992. 97

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the ACM Symposium on Volume Visualization*, 1994. 29

[Cha96] M. Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *IEEE Visualization*, pages 127–132, 1996. 97

[Che73] H. Chernoff. The use of faces to represent points in k-dimensional space graphically. *Journal of American Statistical Association*, (68):361–368, 1973. 16, 96

[Chi] UCSF Chimera. `http://www.cgl.ucsf.edu/chimera/`. 37

[CHWS88] J. Callahan, D. Hopkins, M. Weiser, and B. Shneiderman. An empirical comparison of pie vs. linear menus. In *CHI '88: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 95–100, 1988. 127

[CL93] B. Cabral and L. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of SIGGRAPH*, pages 263–270, 1993. 66

[CMN01] E. Chávez, J. L. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools Appl.*, 14(2):113–135, 2001. 97

[CNM83] S. K. Card, A. Newell, and T. P. Moran. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Inc., 1983. 127

[CUD] Nvidia CUDA Zone. `http://www.nvidia.com/object/cuda_home.html`. 28, 119

[DE04] J. Diepstraten and M. Eissele. In-Depth Performance Analyses of

DirectX9 Shading Hardware concerning Pixel Shader and Texture Performance. In *Shader X3*, 2004. 116

[DGH03] H. Doleisch, M. Gasser, and H. Hauser. Interactive feature specification for focus+context visualization of complex simulation data. In *Proceedings of VisSym '03*, pages 239–248, 2003. 100

[DH93] T. Delmarcelle and L. Hesselink. Visualizing second-order tensor fields with hyperstreamlines. *IEEE Computer Graphics and Applications*, 13(4):25–33, 1993. 80, 81

[dHKP05] G. de Haan, M. Koutek, and F. H. Post. IntenSelect: Using Dynamic Object Rating for Assisting 3D Object Selection. In E. K. R. Blach, editor, *Eurographics Workshop on Virtual Environments (EGVE)*, 2005. 122

[DKL98] E. B. Dam, M. Koch, and M. Lillholm. Quaternions, interpolation and animation. Technical report, Department of Computer Science, University of Copenhagen, 1998. 64

[dmx] Distributed Multihead X Project. `http://dmx.sourceforge.net/`. 130

[Don05] W. Donnelly. *GPU Gems 2*, chapter Per-Pixel Displacement Mapping with Distance Functions. Addison-Wesley, 2005. 85

[dRFK+05] A. del Río, J. Fischer, M. Köbele, D. Bartz, and W. Straßer. Augmented Reality Interaction for Semiautomatic Volume Classification. In E. K. R. Blach, editor, *Eurographics Workshop on Virtual Environments (EGVE)*, pages 113–120, 2005. 123

[DV06] G. Dodson and C. S. Verma. Protein flexibility: its role in structure and mechanism revealed by molecular simulations. *Cell Mol Life Sci*, 63(2):207–219, 2006. 36

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, pages 9–16, 2001. 29, 102

[ESM+05] F. Enders, N. Sauber, D. Merhof, P. Hastreiter, C. Nimsky, and M. Stamminger. Visualization of white matter tracts with wrapped streamlines. In *Proceedings of IEEE Visualization 2005*, pages 51–58, 2005. 80

[FL95] C. Faloutsos and K.-I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 163–174, 1995. 18, 97

[FS93] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual

environments. In *Proceedings of SIGGRAPH '93*, pages 247–254, 1993. 39

[Fur86] G. W. Furnas. Generalized fisheye views. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23, 1986. 141

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990. 24

[GP03] G. Guennebaud and M. Paulin. Efficient screen space approach for Hardware Accelerated Surfel Rendering . In *Proceedings of Vision, Modeling and Visualization '03*, pages 485–495, 2003. 34

[gpg] SIGGRAPH 2004 GPGPU Workshop `http://www.gpgpu.org/s2004/`. 108

[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*, pages 65–72, 2009. 20, 33

[Gro05] S. Grottel. Visualization of clustering fluctuations over time. Master's thesis, Universität Stuttgart, 2005. 68

[GRVE07] S. Grottel, G. Reina, J. Vrabec, and T. Ertl. Visual Verification and Analysis of Cluster Detection for Molecular Dynamics. In *Proceedings of IEEE Visualization '07*, pages 1624–1631, 2007. 20, 33

[GTK] the GIMP Toolkit. `http://www.gtk.org/`. 130

[Gum03] S. Gumhold. Splatting illuminated ellipsoids with depth correction. In *VMV*, pages 245–252, 2003. 18, 34, 39

[Hab90] R. B. Haber. Visualization techniques for engineering mechanics. *Computing Systems in Engineering*, 1(1):37–50, 1990. 80

[Har96] J. C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996. 81, 85

[HBC+92] T. T. Hewett, R. Baecker, S. Card, T. Carey, J. Gasen, M. Mantei, G. Perlman, G. Strong, and W. Verplank. Acm sigchi curricula for human-computer interaction. Technical report, 1992. 30

[HE03] M. Hopf and T. Ertl. Hierarchical Splatting of Scattered Data. In *Proceedings of IEEE Visualization '03*. IEEE, 2003. 23, 40, 53, 69, 123

[Hea96] C. G. Healey. Choosing effective colours for data visualization. In *Proceedings of IEEE Visualization '96*, pages 263–270, 1996. 33

[Hee92] C. Heeter. Being there: the subjective experience of presence. *Presence: Teleoper. Virtual Environ.*, 1(2):262–271, 1992. 121

[HHS93] H.-C. Hege, T. Höllerer, and D. Stalling. Volume rendering - mathematical models and algorithmic aspects. ZIB Technical Re-

port 93-7, 1993. 28

[Hil55] T. L. Hill. Molecular clusters in imperfect gases. *The Journal of Chemical Physics*, 23:617–622, April 1955. 77

[HJ05] C. D. Hansen and C. R. Johnson, editors. *The Visualization Handbook*. Elsevier, 2005. 57

[HLE04] M. Hopf, M. Luttenberger, and T. Ertl. Hierarchical Splatting of Scattered 4D Data. *IEEE Computer Graphics and Applications*, 24(4):64–72, 2004. 70

[HM03] H. Hauser and M. Mlejnek. Interactive volume visualization of complex flow semantics. In *Proceedings Workshop Vision, Modelling, and Visualization 2003 (VMV'03)*, 2003. 99

[HOF05] A. Halm, L. Offen, and D. Fellner. BioBrowser: A Framework for Fast Protein Visualization. In *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization 2005*, 2005. 38

[IG98] V. Interrante and C. Grosch. Visualizing 3D flow. *IEEE Computer Graphics and Applications*, 18(4):49–53, 1998. 66

[Ins85] A. Inselberg. The plane with parallel coordinates. *The Visual Computer*, (1):69–91, 1985. 96

[JEH02] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Lagrangian-Eulerian Advection of Noise and Dye Textures for Unsteady Flow Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):211–222, 2002. 66

[Jmo] Jmol Applet. `http://jmol.sourceforge.net/`. 37, 38

[JWH+04] Y. Jang, M. Weiler, M. Hopf, J. Huang, D. S. Ebert, K. P. Gaither, and T. Ertl. Interactively Visualizing Procedurally Encoded Scalar Fields. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '04*, 2004. 23

[KAK95] D. A. Keim, M. Ankerst, and H.-P. Kriegel. Recursive pattern: A technique for visualizing very large amounts of data. In *Proceedings of IEEE Visualization '95*, page 279, 1995. 96

[Kan01] E. Kandogan. Visualizing multi-dimensional clusters, trends, and outliers using star coordinates. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 107–116, 2001. 96

[KBH04] R. Kosara, F. Bendix, and H. Hauser. Timehistograms for large, time-dependent data. In *IEEE TCVG Symposium on Visualization VisSym '04*, pages 45–54, 2004. 67

[KBR06] J. Kessenich, D. Baldwin, and R. Rost. The opengl shading language. `http://www.opengl.org/documentation/glsl/`, 2006. 24, 27

[KE04] T. Klein and T. Ertl. Illustrating Magnetic Field Lines using a Discrete Particle Model. In *Workshop on Vision, Modelling, and Visualization VMV '04*, 2004. 34, 39, 62

[Kei00] D. A. Keim. Designing pixel-oriented visualization techniques: Theory and applications. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):59–78, 2000. 95

[Kei02] D. A. Keim. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):1–8, 2002. 71

[KHI+03] R. Kosara, C. G. Healey, V. Interrante, D. H. Laidlaw, and C. Ware. User studies: Why, how, and when. *Computer Graphics and Applications*, 23(4):20–25, 2003. 134

[Kie06] R. Kieble. Clustering algorithms to detect nucleation in gases. Master's thesis, Universität Stuttgart, 2006. 77

[KiN] Kinemage, Next Generation. `http://kinemage.biochem.duke.edu/software/king.php`. 38

[Kin04a] G. Kindlmann. Superquadric tensor glyphs. In *Proceedings Eurographics - IEEE TCVG Symposium on Visualization*, pages 147–154, 2004. 80

[Kin04b] G. Kindlmann. *Visualization and Analysis of Diffusion Tensor Fields*. PhD thesis, School of Computing, University of Utah, 2004. 80, 81

[KSH03] R. Kosara, G. N. Sahling, and H. Hauser. Interactive poster: Linking scientific and information visualization with interactive 3D scatterplots. In *Poster Compendium of VIS '03*, pages 96–97, 2003. 100

[KW99] G. Kindlmann and D. Weinstein. Hue-balls and lit-tensors for direct volume rendering of diffusion tensor fields. In *Proceedings of IEEE Visualization '99*, pages 183–189, 1999. 79

[KW03a] J. Krueger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of IEEE Visualization 2003*, 2003. 109

[KW03b] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003. 27

[KWH00] G. Kindlmann, D. Weinstein, and D. Hart. Strategies for direct volume rendering of diffusion tensor fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):124–138, 2000. 79

[Lak04] A. Lakhia. Efficient interactive rendering of detailed models with hierarchical levels of detail. In *Proceedings of the International*

Symposium on 3D Data Processing, Visualization, and Transmission, pages 275–282, 2004. 39

[LAS+02] N. Lori, E. Akbudak, J. Shimony, T. Cull, A. Snyder, R. Guillory, and T. Conturo. Diffusion tensor fiber tracking of human brain connectivity: aquisition methods, reliability analysis and biological results. *NMR in Biomedicine*, 15(7-8):494–515, 2002. 80

[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*, 12(5):1053–1060, 2006. 71

[LC87] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987. 28

[Lev88] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988. 28, 29

[LG94] J. Liang and M. Green. JDCAD: A highly interactive 3D modeling system. *Computers & Graphics*, 18(4):499–506, 1994. 122

[LH04] P. Lacz and J. C. Hart. Procedural Geometric Synthesis on the GPU. In *Proceedings of the GP^2 Workshop*, 2004. 39

[Lin68] A. Lindenmayer. Mathematical models for cellular interaction in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18:280–289, 1968. 39

[LJ31] J. E. Lennard-Jones. Cohesion. In *Proceedings of the Physical Society*, number 43, pages 461–482, 1931. 36

[LKS+06] A. Lefohn, J. M. Kniss, R. Strzodka, S. Sengupta, and J. D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006. 27

[LPC+00] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project: 3D scanning of large statues. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 131–144. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000. 34

[LSM91] X. Lin, D. Soergel, and G. Marchionini. A self-organizing semantic map for information retrieval. In *SIGIR '91: Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 262–269, 1991. 97

[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*, 13(6):1616–1623,

November/December 2007. 56

[LWW90] J. LeBlanc, M. O. Ward, and N. Wittels. Exploring n-dimensional databases. In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pages 230–237, 1990. 96

[Mar02] E. Martz. Protein explorer: Easy yet powerful macromolecular visualization. *Trends in Biochemical Sciences*, 27:107–109, 2002. 37

[Max04] N. Max. Hierarchical molecular modelling with ellipsoids. *Journal of Molecular Graphics and Modelling*, 23, 2004. 38

[MBDM97] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. Infinitereality: a real-time graphics system. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 293–302, 1997. 23

[MC03] A. Morrison and M. Chalmers. Improving hybrid mds with pivot-based searching. In *Proceedings of IEEE Information Visualization '03*, 2003. 97

[MGA03] W. R. Mark, S. Glanville, and K. Akeley. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proc. of SIGGRAPH '03*, pages 896–907, 2003. 24, 27

[MGB+05] J. L. Moreland, A. Gramada, O. V. Buzko, Q. Zhang, and P. E. Bourne. The molecular biology toolkit (mbt): a modular platform for developing molecular visualization applications. *BMC Bioinformatics*, 6(21), 2005. 38

[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*, 19(11):791–800, 2005. 37

[Mil56] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956. 30, 128, 133

[MP03] R. Měch and P. Prusinkiewicz. Generating subdivision curves with L-systems on a GPU. In *Proceedings of SIGGRAPH '03*, pages 1–1, 2003. 39

[MPZ+02] W. Matusik, H. Pfister, R. Ziegler, A. Ngan, and L. McMillan. Acquisition and rendering of transparent and refractive objects. In *Proceedings of Eurographics '02*, pages 19–24, 2002. 34

[MQP02] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, 2002. 28, 119

[MRC02] A. Morrison, G. Ross, and M. Chalmers. A hybrid layout algo-

rithm for sub-quadratic multidimensional scaling. In *IEEE Information Visualization '02*, pages 152–158, 2002. 97

[MRIB00] E. R. Melhema, L. J. Ryuta Itoha, and P. B. Barkera. Diffusion tensor mr imaging of the brain: Effect of diffusion weighting on trace and anisotropy measurements. *American Journal of Neuroradiology*, 21:1813–1820, 2000. 79

[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*, 1(5):39–47, 1999. 39

[Nor88] D. A. Norman. *The Design of Everyday Things*. MIT Press, 1988. 9, 30, 122, 131

[NVS] NVShaderPerf profiling tool. `http://developer.nvidia.com/object/nvshaderperf_home.html`. 92

[OIEE01] F. Oellien, W. D. Ihlenfeldt, K. Engel, and T. Ertl. Multi-variate interactive visualization of data from digital laboratory notebooks. In *ECDL: Workshop Generalized Documents*, 2001. 99

[Opea] OpenCL Specification and Registry. `http://www.khronos.org/registry/cl`. 119

[Opeb] The OpenGL 2.1 Specification. `http://www.opengl.org/documentation/specs/version2.1/glspec21.pdf`. 24

[Opec] OpenGL3.0 BOF SIGGRAPH 2007 presentations. `http://www.khronos.org/library/detail/siggraph_2007_opengl_birds_of_a_feather_bof_presentation/`. 45

[PAB02] S. Pajevic, A. Aldroubi, and P. Basser. A continuous tensor field approximation of discrete dt-mri data for extracting microstructural and architectural features of tissue. *Journal of Magnetic Resonance*, 154(1):85–100, 2002. 79

[pbu] OpenGL ARB pbuffer specification, `http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pbuffer.txt`. 110

[PD84] T. Porter and T. Duff. Compositing digital images. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 253–259, 1984. 26

[PG88] R. Pickett and G. Grinstein. Iconographics display for visualizing multidimensional data. In *IEEE SMC '98*, pages 514–519, 1988. 16, 96

[Pha] The PHANTOM Desktop device home page. `http://www.sensable.com/haptic-phantom-desktop.htm`. 30

[PP99] S. Pajevic and C. Pierpaoli. Color schemes to represent the ori-

entation of anisotropic tissues from diffusion tensor data: Application to white matter fiber tract mapping in the human brain. *Magnetic Resonance in Medicine*, 42(3):526–540, 1999. 79

[PPWS95] F. H. Post, F. J. Post, T. V. Walsum, and D. Silver. Iconic techniques for feature visualization. In *Proceedings of the 6th conference on Visualization '95*, page 288, 1995. 33

[PvW94] F. H. Post and J. J. van Wijk. Visual representation of vector fields - recent developments and research directions. In L. Rosenblum, R. A. Earnshaw, J. Encarnacao, and H. Hagen, editors, *Scientific Visualization: Advances and Challenges*, pages 367–390, 1994. 34

[Pym] PyMOL. `http://pymol.sourceforge.net/`. 37

[PZvBG00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *Proceedings of SIGGRAPH '00*, pages 335–342, 2000. 34

[Rag02] J. Ragas. Interactive visualization of flowfields in an immersive virtual environment, 2002. 33

[RBE+06] G. Reina, K. Bidmon, F. Enders, P. Hastreiter, and T. Ertl. GPU-Based Hyperstreamlines for Diffusion Tensor Imaging. In *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization 2006*, pages 35–42, 2006. 20, 33

[RE04] G. Reina and T. Ertl. Volume visualization and visual queries for large high-dimensional datasets. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '04*, pages 255–260, 2004. 95

[RE05a] G. Reina and T. Ertl. Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization. In *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization 2005*, 2005. 33

[RE05b] G. Reina and T. Ertl. Implementing FastMap on the GPU: Considerations on General-Purpose Computation on Graphics Hardware. In *Theory and Practice of Computer Graphics '05*, pages 51–58, 2005. 95

[REE+94] L. Rosenblum, R. Earnshaw, J. Encarnacao, H. Hagen, A. Kaufman, S. Klimenko, G. Nielson, F. Post, and D. Thalmann. *Scientific Visualization: Advances and Challenges*. Academic Press, 1994. 21

[RKE07] G. Reina, T. Klein, and T. Ertl. *Point-Based Graphics*, chapter Visualization of Attributed 3D Point Datasets, pages 420–435. Morgan Kaufmann Publishers, 2007. 20

[RL00] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, pages 343–352, 2000. 33, 40

[RLLEE03] G. Reina, S. Lange-Last, K. Engel, and T. Ertl. Guided Navigation in Task-Oriented 3D Graph Visualizations. In *Theory and Practice of Computer Graphics '03*, pages 26 – 33, 2003. 3, 20, 142, 145

[Rob92] W. Robinett. Synthetic experience: a proposed taxonomy. *Presence: Teleoper. Virtual Environ.*, 1(2):229–247, 1992. 121

[RRE06] A. Rosiuta, G. Reina, and T. Ertl. Flexible Interaction with Large Point-Based Datasets. In *Theory and Practice of Computer Graphics '06*, 2006. 20, 121

[RSBE01] S. Roettger, M. Schulz, W. Bartelheimer, and T. Ertl. Automotive Soiling Simulation Based On Massive Particle Tracing. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 309–317,363, 2001. 66

[RSEB+00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00*, pages 109–118,147. Addison-Wesley Publishing Company, Inc., 2000. 29

[RSR+03] D. Rose, S. Stegmaier, G. Reina, D. Weiskopf, and T. Ertl. Non-invasive Adaptation of Black-box User Interfaces. In *Proceedings of Fourth Australasian User Interface Conference AUIC 2003*, pages 19–24, 2003. 3, 20

[SFB94] M. Stone, K. Fishkin, and E. Bier. The Movable Filter as a User Interface Tool. In *Proceedings CHI '94*, pages 306–312, 1994. 101

[SFCN02] Y. Suzuki, I. Fujishiro, L. Chen, and H. Nakamura. Case Study: Hardware-Accelerated Selective LIC Volume Rendering. In *Proceedings of IEEE Visualization*, pages 485–488, 2002. 66

[SFGF72] J. Siegel, E. Farrell, R. Goldwyn, and H. Friedman. The surgical implication of physiologic patterns in myocardial infarction shock. *Surgery*, 72:126–141, 1972. 96

[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 '97*, pages 77–86, 1997. 62

[SGS01] J. E. Stone, J. Gullingsrud, and K. Schulten. A system for interactive molecular dynamics simulation. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 191–194, 2001. 39

[SGS05] C. Stoll, S. Gumhold, and H. Seidel. Visualization with stylized line primitives. In *Proceedings of IEEE Visualization '05*. IEEE, 2005. 81

[SMW95] R. A. Sayle and E. J. Milner-White. RASMOL: biomolecular

graphics for all. *Trends in Biochemical Sciences*, 20(9):374–376, 1995. 37

[SP92] B. Shneiderman and C. Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 1992. 30

[SSKE05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 187–195, 2005. 29

[ST90] P. Shirley and A. A. Tuchman. Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, volume 24, pages 63–70, 1990. 29

[STDS95] B. Spence, L. Tweedie, H. Dawkes, and H. Su. Visualization for functional design. In *INFOVIS '95: Proceedings of the 1995 IEEE Symposium on Information Visualization*, 1995. 23

[SWPG05] F. Sadlo, T. Weyrich, R. Peikert, and M. Gross. A practical structured light acquisition system for point-based geometry and texture. In *Proceedings of the Symposium on Point-Based Graphics '05*, pages 89–98, 2005. 34

[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*, 12(5):1237–1244, 2006. 61

[TL04] R. Toledo and B. Lévy. Extending the graphic pipeline with new gpu-accelerated primitives. Tech report, INRIA Lorraine, 2004. 35

[UBBvZ96] A. Ulug, O. Bakht, R. Bryan, and P. van Zijl. Mapping of human brain fibers using diffusion tensor imaging. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, page 1325, 1996. 79

[VMD] Visual Molecular Dynamics. `http://www.ks.uiuc.edu/Research/vmd/`. 37

[vWPSP96] T. van Walsum, F. H. Post, D. Silver, and F. J. Post. Feature extraction and iconic visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):111–119, 1996. 80

[vWvL93] J. J. van Wijk and R. van Liere. Hyperslice: visualization of scalar functions of many variables. In *VIS '93: Proceedings of the 4th conference on Visualization '93*, pages 119–125, 1993. 23

[WDC99] K. Watsen, R. Darken, and M. Capps. A handheld computer as

an interaction device to a virtual environment. In *Proceedings of the third Immersive Projection Technology Workshop*, 1999. 123

[WE02] D. Weiskopf and T. Ertl. A Depth-Cueing Scheme Based on Linear Transformations in Tristimulus Space. Technical Report TR-2002/08, Universität Stuttgart, Fakultät Informatik, September 2002. 60, 111

[WE04] D. Weiskopf and T. Ertl. A Hybrid Physical/Device-Space Approach for Spatio-Temporally Coherent Interactive Texture Advection on Curved Surfaces. In *Proceedings of Graphics Interface 2004*, pages 263–270, 2004. 66

[Web] WebMol Applet. `http://www.cmpharm.ucsf.edu/~walther/webmol.html`. 38

[Wei05] M. Weiler. *Hardware-beschleunigte Volumenvisualisierung auf adaptiven Datenstrukturen*. PhD thesis, Universität Stuttgart, 2005. 143

[Wer23] M. Wertheimer. Untersuchungen zur lehre von der gestalt ii. *Psychologische Forschung*, 4:301–350, 1923. 141

[Wes90] L. Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, 1990. 29

[WFP+01] M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Straßer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *Proceedings of SIGGRAPH 2001*, pages 361–370, 2001. 35

[WG95] M. M. Wloka and E. Greenfield. The virtual tricorder: A uniform interface for virtual reality. In *ACM Symposium on User Interface Software and Technology*, pages 39–40, 1995. 123

[Wil96] G. Wills. Selection: 524,288 ways to say "this is interesting". In *Proceedings InfoVis*, pages 54–60, 1996. 122

[WL01] B. Wuensche and R. Lobb. The visualization of diffusion tensor fields in the brain. In *Proc. of the International Conference on Mathematics and Engineering Techniques in Medicine and Biological Science, METMBS*, pages 498–504, 2001. 80

[WMK+99] C.-F. Westin, S. E. Maier, B. Khidhir, P. Everett, F. A. Jolesz, and R. Kikinis. Image processing for diffusion tensor magnetic resonance imaging. In *Medical Image Computing and Computer-Assisted Intervention*, pages 441–452, 1999. 90

[WMM+02] C.-F. Westin, S. E. Maier, H. Mamata, A. Nabavi, F. A. Jolesz, and R. Kikinis. Processing and visualization of diffusion tensor MRI. *Medical Image Analysis*, 6(2):93–108, 2002. 79, 80

[WS82] G. Wyzecki and W. S. Stiles. *Color Science: Concepts and Methods,*

Quantitative Data and Formulae. Wiley, 2nd edition, 1982. 30

[WS02] M. Wand and W. Straßer. Multi-resolution rendering of complex animated scenes. *Computer Graphics Forum*, 21(3):483–491, 2002. 35

[WSE05] D. Weiskopf, T. Schafhitzel, and T. Ertl. Real-Time Advection and Volumetric Illumination for the Visualization of 3D Unsteady Flow. In *Proceedings of EG/IEEE TCVG Symposium on Visualization Eurovis '05*, pages 13–20, 2005. 66

[WTP+95] J. A. Wise, J. J. Thomas, K. Pennock, D. Lantrip, M. Pottier, A. Schur, and V. Crow. Visualizing the non-visual: spatial analysis and interaction with information from text documents. In *Proceedings of the 1995 IEEE Symposium on Information Visualization*, pages 51–58, 1995. 97

[ZDL03] S. Zhang, C. Demiralp, and D. H. Laidlaw. Visualizing diffusion tensor mr images using streamtubes and streamsurfaces. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):454–462, 2003. 80

[ZPvBG01] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proceedings of SIGGRAPH 2001*, pages 371–378, 2001. 34

[Zwi95] D. Zwillinger, editor. *Standard Mathematical Tables and Formulas*. CRC Press, 30th edition, 1995. 35