Proceedings Of
The 6^{th} High-End Visualization Workshop

Dec. 8^{th} to 12^{th}, Obergurgl, Austria

Werner Benger, Andreas Gerndt, Simon Su, Wolfram Schoor,
Michael Koppitz, Wolfgang Kapferer, Hans-Peter Bischof,
and Massimo Di Pierro
(Editors)

Bibliographische Information der Deutschen Bibliothek:
Die deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Werner Benger, *CCT/LSU, Baton Rouge, Louisiana, USA*
Andreas Gerndt, *German Aerospace Center, Braunschweig, Germany*
Simon Su, *NOAA, New Jersey, USA*
Wolfram Schoor, *EADS Deutschland GmbH, Manching, Germany*
Michael Koppitz, *MPI for Gravitational Physics, Potsdam, Germany*
Wolfgang Kapferer, *Institute for Astro- and Particle Physics, Innsbruck, Austria*
Hans-Peter Bischof, *Rochester Institute of Technology, Rochester, NY, USA*
and Massimo Di Pierro, *University of DePaul, USA*
(Editors)

Proceedings of the 6^{th} High-End Visualization Workshop

Dec. 8^{th} to 12^{th}, Obergurgl, Austria

Hardenbergstraße 5 · 10623 Berlin
http://www.lehmanns.de/

ISBN 978-3-86541-361-1

Druck und Verarbeitung: Docupoint Magdeburg, www.docupoint-md.de

Cover page provided by Sebastian Friedrich.

Preface

The *High-End-Visualization Workshop* is a tentatively annual meeting of active researchers in the field of scientific visualization with application domain scientists, in particular from high-performance computing. Historically it originated from a creative workshop involving members of the Numerical Relativity Group at the Max-Planck-Institute for Gravitational Physics in Potsdam, Germany, and researchers of the Department for Scientific Visualization at the Zuse-Institute Berlin, Germany, as well as astrophysicists from the Institute for Astrophysics at the University of Innsbruck, Austria. Since its original birth in 2004 the workshop has grown its own momentum. Each year a special focus area is featured:

- 1^{st} High End Visualization Workshop (Austria, 2004): *Numerical Relativity Mesh Refinement Visualization Meeting* ,
- 2^{nd} High End Visualization Workshop (Austria, 2005): *Multipatch Methods*,
- 3^{rd} High End Visualization Workshop (Austria, 2006): *Public Relations and Public Outreach of Scientific Visualization*,
- 4^{th} High End Visualization Workshop (Austria, 2007): *Visualization of Non-Trivial Data Structures* ((ISBN 978-3-86541-216-4) - appendix B.1,
- 5^{th} High End Visualization Workshop (Louisiana, 2009): *Remote and Collaborative Visualization*(ISBN 978-3-86541-330-7) - appendix B.2,
- 6^{th} High End Visualization Workshop (Austria, 2010): *Design and Concept of Visualization Frameworks* (ISBN 978-3-86541-361-1).

The 6^{th} High End Visualization Workshop has received ten paper submissions, nine of which have been accepted for publication, three of those conditionally. Each contribution was evaluated by at least three independent reviewers. Special thanks go to the Markus Haider, Sabine Schindler, Gabrielle Allen and Susanne Brenner for support of the workshop.

Werner Benger and the Organizers, November 2010

List of Contributions

Article 1

FSSteering: A Distributed Framework for Computational Steering in a Script-based CFD Simulation Environment

Christian Wagner[1,2], Markus Flatken[1], Michael Meinel[1],
Andreas Gerndt[1], Hans Hagen[2]
[1]German Aerospace Center, Braunschweig, Germany
[2]University of Kaiserslautern, Germany
christian.wagner@dlr.de, markus.flatken@dlr.de,
michael.meinel@dlr.de, andreas.gerndt@dlr.de,
hagen@informatik.uni-kl.de

In order to get insight into interesting flow phenomena, the traditional work-flow of computational fluid dynamics (CFD) consists of setting up and computing the flow field followed by a consecutive post-processing analysis. Only after this analysis one can identify parameters that may have been set wrongly in a configuration stage. Once these parameters are corrected, another time-consuming loop has to be started. To identify inadequate parameter settings already during the simulation run, online monitoring concepts were introduced. Combined with computational steering methods, parameter values can additionally be adjusted which eventually reduces the number of required iterations to yield satisfactory results.

At the German Aerospace Center, a comprehensive framework called FlowSimulator has been developed to offer a generic Python-based interface for the management of CFD simulations. It can easily be en-

hanced by add-ons. One of these extensions is *FSSteering* which is described in this paper in more detail. As a computational steering environment, *FSSteering* provides functionalities essential for interactive visualization and explorative analysis. Besides existing computational steering environments and frameworks, a user-centred and domain-specific view is proposed. Existing functionality can be reused without rewriting simulation code to enable for effective steering in CFD.

To be more efficient, components of the architecture are distributed across different resources. Whereas the CFD simulation typically runs on a parallel supercomputer, the visualization is carried out on a high-performance virtual reality system which allows interactive data exploration. The post-processing in between can be performed on the supercomputer or on a separate parallelization cluster. But it is also possible to switch between different existing post-processing toolkits. This is just possible because of the very flexible configuration management of the distributed steering framework. We will demonstrate the steering capabilities and the system flexibility by two current research examples. An outlook for future steps concludes this paper.

1.1 Introduction

In order to gain insights into complex flow situations, computational simulation is a common tool for modern engineers. Therefore, a computational fluid dynamics simulation is set up involving necessary parameters. After solving by a cluster or supercomputer post-processing algorithms are applied to generate visual feedback. Many parameters chosen wrong can only be identified at that point and potentially the simulation has to be done again with tweaked parameters. With iteration times of days or even weeks methods with higher productivity are desirable for providing quick research insights. Therefore, being able to check if a simulation was setup properly and is still on track is important. If possible, changes to guide the simulation should be applied during runtime.

To tackle this situation, computational steering systems were developed to interact with ongoing simulation runs. Most of the computational steering environments available are enhanced visualization tools and concentrate on data management providing meaningful visualizations. Complicated instrumentation of simulation codes is needed to make data available to those systems. Contrary, computational steering frameworks concentrate on easy simulation coupling with minimalistic interfaces. Visualization and analysis have to be implemented by the user. However, main drawback of both approaches is that all steerable parameters as well as callable methods have to be known at compile time.

In this paper, we introduce a domain-specific approach heavily depending on an existing scripting environment. We developed the computational steering environment *FSSteering* as an extension to the German Aerospace Center's computational fluid dynamics system FlowSimulator. For this reason, simulation scripts can be made steerable without internal knowledge by users. Nearly no instrumentation and data conversion is needed. Furthermore, steering commands to be executed are mostly interpreted by the *FSSteering* extension. Therefore, domain-specific tasks provided by the FlowSimulator environment can be executed or parameters can be changed during runtime without either being known to the simulation script or having to be implemented by CFD engineers . For example, the underlying mesh of any CFD simulation can be changed and adapted during runtime resulting in better simulation convergence without changes to the simulation setup or script.

Since CFD data are multi-modal with complex features, we coupled our systems with VRFlowVis, an explorative visualization systems including a parallel post-processing system.

The remaining paper is structured as follows: In the next section, related work in the field of computational steering environments and frameworks is reflected including an overview of FlowSimulator and VRFlowVis. Then, the developed *FSSteering* architecture is presented. In section 1.4, two steering examples are shown, followed by final conclusion and future work.

1.2 Related Work

Since steering simulations is of interest for many years now, a lot of work has been done. An overview of earlier systems can be found in [Mulder et al., 1998]. Online monitoring is essential to identify in what kind a simulation has to be steered. Therefore, most steering systems concentrate on visualization or are even enhanced visualization tools, like [Parker et al., 1997] and [Eickermann et al., 2005]. Native frameworks like [Jenz & Bernreuther, 2010] are available to enable for computational steering, but having high adaptation overhead to specific problems. [Coulaud et al., 2003] uses XML descriptions of simulation scripts to handle data and concurrency at instrumentation points. Only few existing systems try to tackle domain-specific requirements, one CFD-specific is [Kreylos et al., 2002].

The FlowSimulator is an open and efficient framework to unify massively parallel and multidisciplinary CFD simulations independantly from the tools incorporated [Meinel & Einarsson, 2010]. This is achieved by a layered approach. The FlowSimulator DataManager (FSDM) forms the common backbone and provides a common interface to store and exchange data in memory. Written in C++ it provides a number of classes that hold structured data typical for CFD-related numerical simulations. Using the automatic interface generator SWIG [Beazley,

1996] all of FSDM's interfaces are also provided in Python.

Explorative and interactive visualization is supported using the VRFlowVis application, a visualization frontend for steady and unsteady CFD data sets based on ViSTA and ViSTA FlowLib [Schirski et al., 2003]. ViSTA allows the frontend to scale from simple desktop systems to high-end immersive VR environments. ViSTA FlowLib is a specialized library that provides particular interaction methods [Wolter et al., 2007a][Wolter et al., 2006] and efficient rendering techniques for working with time-dependent CFD data.

The rendered features are extracted from the raw data and mapped to visualization components by a post-processing application based on Viracocha [Gerndt et al., 2004][Wolter et al., 2007b]. It is decoupled from the visualization frontend and distributed to High Performance Computing (HPC) resources, preferably the same resource used by the simulation to be steered. Visualization features are extracted in parallel, and as soon as first results are available the extracted geometry data is sent back to VRFlowVis to be rendered.

1.3 Computational Steering Architecture

The developed computational steering architecture aims on enabling computational fluid dynamics simulations to be steered with little impact on already existing simulation scripts. Therefore, *FSSteering* was developed as an extension to the FlowSimulator system providing easy access to existing functionality and coupling simulation scripts with parallel post-processing back-end as well as front-end systems.

In the target work-flow different connected computing systems are involved, c.f. figure 1.1. A supercomputer or cluster system is supporting a set of simulation tasks in batch-processing. To steer one of the running simulations on-demand different front-end and back-end systems need to be attached in a flexible connection topology dealing with heterogeneous networks.

1.3.1 System Architecture

The overall *FSSteering*-architecture can be seen in figure 1.2.

Although *FSSteering* makes use of the scripting interface offered by FlowSimulator, performance-critical tasks need to be implemented efficiently. For this reason, a core module provides connection handling and data transfer methods. Access to these functions is provided by lightweight APIs. The Python-API is also bound to the FlowSimulator-API allowing inheriting its functionality and providing it to the connected applications via command requesting.

Both, Python- and C++-API, are used in the runtime examples in section 1.4.

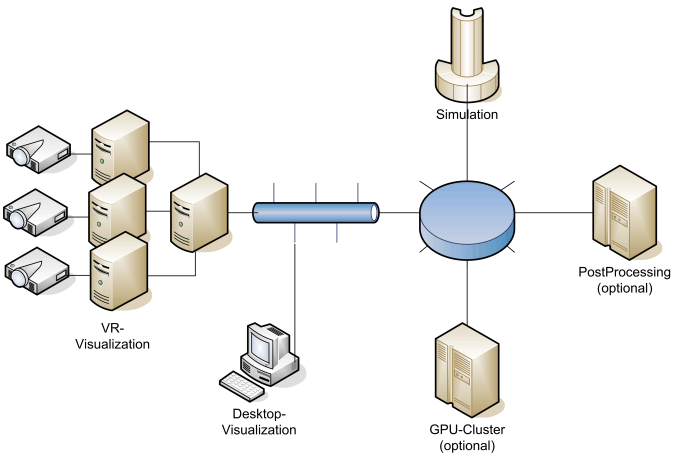


Figure 1.1: CFD simulations can be connected by multiple post-processing and front-end visualization systems on-demand. TCP/IP- as well as MPI-connections can be used for data communication.

1.3.2 Runtime Execution

At runtime a steerable simulations act as servers waiting for connections by clients. Clients again can act as servers allowing arbitrary connection topologies.

A connected client sends commands to the simulation server and waits for response. A set of predefined system commands exists for registration and updating variables and sending geometry or field data over connections. Calling domain-dependent FlowSimulator functionalities like mesh adaptation offer the possibility to change simulation behavior without being implemented in the application scripts in the first place. All commands unknown to *FSSteering* are assumed to be user-commands and are returned to the caller, e.g. the simulation script. For simple handling commands are represented as Python dictionaries including necessary parameters and are mapped to dictionaries of strings in the C++-API. Commands are sent through the system in a serialized representation. The interpretation occurs when triggered by the simulation.

The execution of commands is based on message queues. For command execution with centralized request management [Esnard et al., 2004] a simple, yet efficient synchronization scheme is used. All commands are gathered at a clients master node and are send to the servers master node. When a simulation triggers the processing of upcoming commands, the server broadcasts all new commands to the servers slaves. This choice perfectly fits to the single program multiple data programming model used in FlowSimulator scripts.

Special care was taken managing different connections in order to provide a flexible connection topology. Although command communication is always gathered

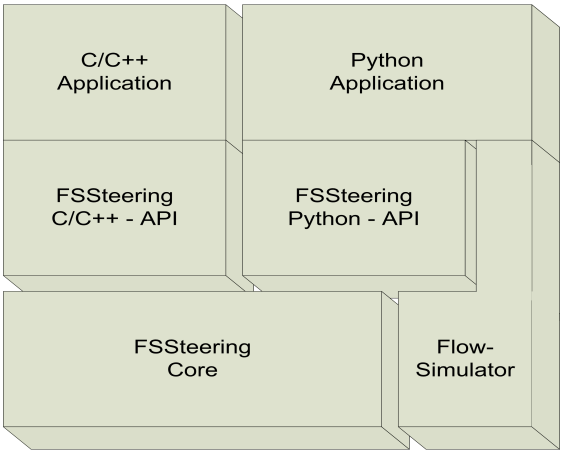
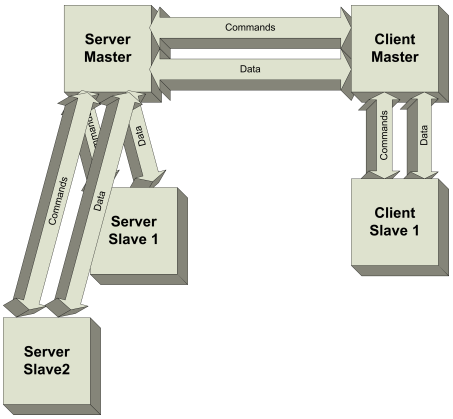
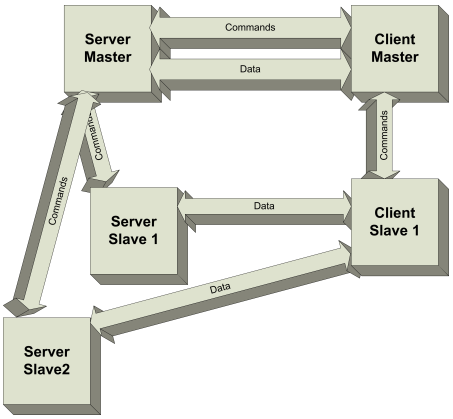


Figure 1.2: *FSSteering*'s main functionalities are implemented in the core module and made accessible by lightweight APIs. The Python-API can also use functions of FlowSimulator.

and scattered through the master nodes this does not hold for data communication. As depicted in figure 1.3, additional to 1-to-1 connection via master to master connection it is possible to establish n:m connections, where each server node is connected to an arbitrary client node. This setting is used in the steering examples of section 1.4. For general purposes, geometry and field data can be sent as raw binary data, the VTK file-format is also supported.



(a) 1:1-connection: Data and commands are gathered at master nodes and redistributed to slave nodes.



(b) n:m-connection: While commands are gathered and redistributed, data is distributed in parallel.

Figure 1.3: Commands are always gathered and redistributed in master nodes. For data, each data node can be connected to arbitrary client nodes.

1.4 Computational Steering Results

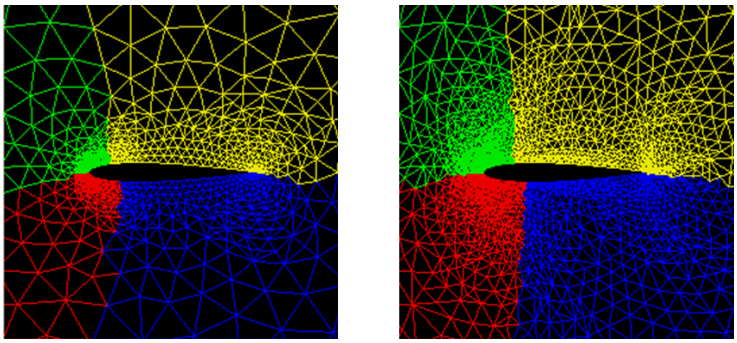
This section demonstrates the effective usage of *FSSteering* in two examples. A FlowSimulator simulation running on four computational nodes is made steerable using the *FSSteering* Python-API. The parallel post-processor Viracocha connects to the computational nodes via parallel data channels, one to each simulation node. Simulation and post-processor are controlled using a ViSTA front-end. In this setup we will show two frequent steering applications.

1.4.1 Numerical Steering

Since the underlying simulation mesh is essential for numerical convergence to physical meaningful results, the additional possibility to influence the mesh during runtime can prevent restarting simulation runs. Figure 1.4 shows the effect of additional adaptation runs initiated in the FlowSimulator environment. Note, that no additional code adjustment are needed since mesh adaptation is one of the algorithms provided in FlowSimulator and *FSSteering*.

1.4.2 Simulation Steering

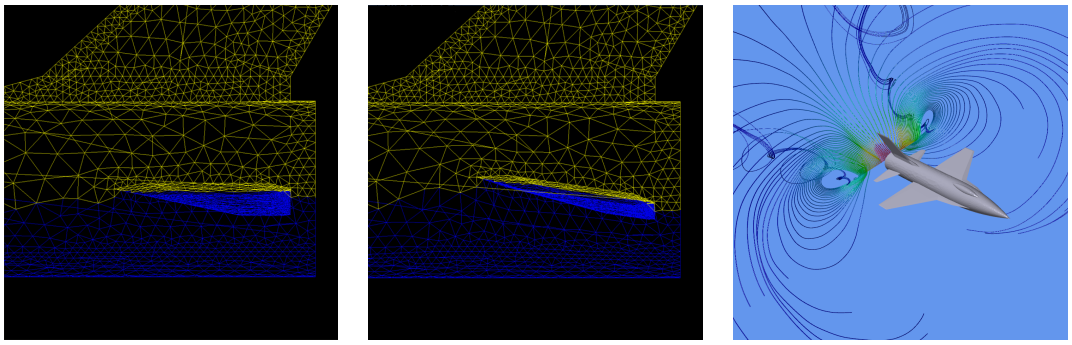
Contrary to the first example, this example shows how a simulation script is enriched with user-defined code, see figure 1.5. The used simulation script has the ability to change aileron, rudder and elevator angles in a synthetic aircraft model.



(a) Initial mesh. (b) Adapted mesh.

Figure 1.4: For an initial computational mesh (a) background adaptation is triggered improving numerical correctness.

FSSteering's abilities to schedule user-defined steering commands during runtime is used to successfully deform the mesh. Mesh deformation is controlled and viewed by the front-end application. Two wire-frame and a virtual reality view of the front-end is shown in figure 1.5.



(a) Initial configuration. (b) Changed angle. (c) Explorative view with changed angle in VR front-end.

Figure 1.5: In simulation steering a command to change elevator angle was triggered, (a) and (b). The influence on the flow field is analyzed in the explorative visualization environment, (c).

1.5 Conclusion and Future Work

In this paper we presented *FSSteering*, a flexible computational steering environment. As an extension to the FlowSimulator framework domain-specific needs of CFD engineers are addressed. A flexible connection and data management between the simulation on the one hand and front-end as well as post-processing back-end modules on the other hand was demonstrated. Existing FlowSimulator functionalities are inherited and can be used with very little programming effort. Simulation-specific functions can be added by users with a convenient Python-API. The combination with an existing parallel post-processor and a virtual-reality environment provides a rich set of analysis and explorative tools.

To serve typical batch processing systems running many simulations at the same time, future work will include management of running simulations to allow for a convenient selection of which simulation to steer. For simulations running with high counts of computational nodes additional data redistribution and streaming methods are needed to provide quick insights. Further research includes investigation of adequate interaction techniques in the front-end visualization to allow best usage of the functionalities provided for CFD simulations steered with *FSSteering*.

Bibliography

[Beazley, 1996] Beazley, D. M. (1996). Swig: an easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4* (pp. 15–15). Berkeley, CA, USA: USENIX Association. URL: `http://portal.acm.org/citation.cfm?id=1267498.1267513`.

[Coulaud et al., 2003] Coulaud, O., Dussere, M., & Esnard, A. (2003). Toward a distributed computational steering environment based on corba.

[Eickermann et al., 2005] Eickermann T., Frings, W., Gibbon, P., Kirtchakova, L., Mallmann, D., & Visser, A. (2005). Steering unicore applications with visit. *Philosophical Transactions of The Royal Society. Journal*.

[Esnard et al., 2004] Esnard, A., Dussere, M., & Coulaud, O. (2004). A time-coherent model for the steering of parallel simulations. In *Europar 2004* (pp. 90–97).: Springer Verlag.

[Gerndt et al., 2004] Gerndt, A., Hentschel, B., Wolter, M., Kuhlen, T., & Bischof, C. (2004). Viracocha: An efficient parallelization framework for large-scale cfd post-processing in virtual environments. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (pp.5̃0). Washington, DC, USA: IEEE Computer Society.

[Jenz & Bernreuther, 2010] Jenz, D. & Bernreuther, M. (2010). The computational steering framework steereo. In *Para 2010*.

[Kreylos et al., 2002] Kreylos, O., Tesdall, A. M., Hamann, B., Hunter, J. K., & Joy, K. I. (2002). Interactive visualization and steering of cfd simulations. In S. Müller & W. Stärzlinger (Eds.), *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002* (pp. 25–34).

[Meinel & Einarsson, 2010] Meinel, M. & Einarsson, G. O. (2010). The flowsimulator framework to unify massively parallel cfd applications. In *Para 2010*.

[Mulder et al., 1998] Mulder, J. D., van Wijk, J., & Liere, R. V. (1998). A survey of computational steering environments. *Future Generation Computer Systems*, 13.

[Parker et al., 1997] Parker, S. G., Weinstein, D. M., & Johnson, C. R. (1997). The scirun computational steering software system.

[Schirski et al., 2003] Schirski, M., Gerndt, A., van Reimersdahl, T., Kuhlen, T., Adomeit, P., Lang, O., Pischinger, S., & Bischof, C. (2003). Vista flowlib - framework for interactive visualization and exploration of unsteady flows in virtual environments. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003* (pp. 77–85). New York, NY, USA: ACM.

[Wolter et al., 2007a] Wolter, M., Bischof, C., & Kuhlen, T. (2007a). Dynamic regions of interest for interactive flow exploration. In *Proceedings of Parallel Graphics and Visualization 2007* (pp. pp. 61–68).

[Wolter et al., 2006] Wolter, M., Hentschel, B., Schirski, M., Gerndt, A., & Kuhlen, T. (2006). Time step prioritising in parallel feature extraction on unsteady simulation data. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization 2006*.

[Wolter et al., 2007b] Wolter, M., Schirski, M., Kuhlen, T., Bischof, C., Bücker, M., Gibbon, P., Joubert, G. R., Mohr, B., (eds, F. P., Wolter, M., Schirski, M., & Kuhlen, T. (2007b). Hybrid parallelization for interactive exploration in virtual environments.

Article 2

Visualization Workflow for Lattice QCD

Brian Schinazi, Yaoqian Zhong, Massimo Di Pierro
School of Computing, College of Computing and Digital Media, DePaul University
243 S Wabash Avenue, Chicago, IL, USA

Vis is a web based application that implements Software as a Service for Lattice QCD computations. Lattice QCD is a numerical approach to the mathematical model that descrbes quarks and gluons, the consituents of protons, neutrons and many other composite particles. Lattice QCD computations are implemented as Markov Chain Monte Carlo. Vis allows to store this data, explore it, schedule computing jobs using a local or remote PBS cluster, and schedule visualization jobs using VisIt as back-end. All the major operations of Vis can be performed via the web-based interface as well as scripted. Vis provides an access control mechanism and strong security features. It is a single platform that may allow physicists to collaborate better by sharing their data online.

2.1 Introduction

Lattice QCD [Massimo Di Pierro, 2006] is a numerical approach to the study of quarks, the elementary constituents of protons, neutrons and other forms of matter. In 1968, the study of physics reached a turning point when the structure and interactions of all known particles were described by a single mathematical expression, known as the Standard Model Lagrangian [Novaes, 1999]. Since that time, predictions based on the Standard Model have been extremely accurate, and have been able to account for the results of every high energy physics experiment.

Still, physicists continue to explore nature at smaller and smaller scales and to look for a breakdown of the model, manifested as a discrepancy between predictions and experiments. This would be a major discovery.

The part of the Standard Model that describes quarks specifically is called Quantum Chromodynamics [Altarelli, 2002]. This constitutes perhaps the most fascinating part of the Standard Model, as quarks are the only elementary particles subject to the strong nuclear force – a highly non-linear interaction that allow quarks to bind together in complex structures. Practically all of the composite particles we see in experiments are made up of quarks.

The goal of Lattice QCD is twofold: to compute from first principles the properties (such as masses and lifetimes) of these composite particles, and to extract fundamental parameters of QCD (such as particles' masses) from a comparison of theory with experiment.

Typical computations consist of taking a small portion of space ($10^{-15}m$ of side) and its evolution over a short period of time ($10^{-23}s$), and then performing a Markov Chain Monte Carlo (MCMC) simulation of all of its possible evolutions (1000 histories). We call the data saved at each MCMC step a gauge configuration. Next, correlation functions are measured over all simulated evolutions of the system. It can be proven that such an algorithm is equivalent to simulating a quantum-mechanical system. Finally, observable quantities are extracted from the correlation functions.

Until recently, visualization techniques have not been used in the study of QCD. The main reason is that the objects being computed have no obvious correspondence with physical 3D objects. The content of the portion of space which is simulated contains purely random data, since each data set is just a step of a MCMC. The physics is encoded in the probability distribution used to generate the MCMC, and not in the data itself.

We believe that there are some useful applications of visualization techniques to Lattice QCD: they can be used for didactic purposes, they can be used to better understand the behavior of the MCMC algorithms, and they can be used to detect certain types of error in the computation.

One type of error we are interested in is a systematic one: the possible long auto-correlation of topological charge distribution. The portion of space-time that is simulated contains a field that can be thought of as the chromo-electro-magnetic field of gluons, the particles that mediate interactions between quarks (analogous to the electro-magnetic field being represented by photons, but in this case having three types of charges). One can also associate a local topological charge density to the field. It should be noted that these fields all live in 4D and therefore must be projected on to 3D in order to be visualized.

The elementary steps of Lattice QCD algorithms are local and, for typical

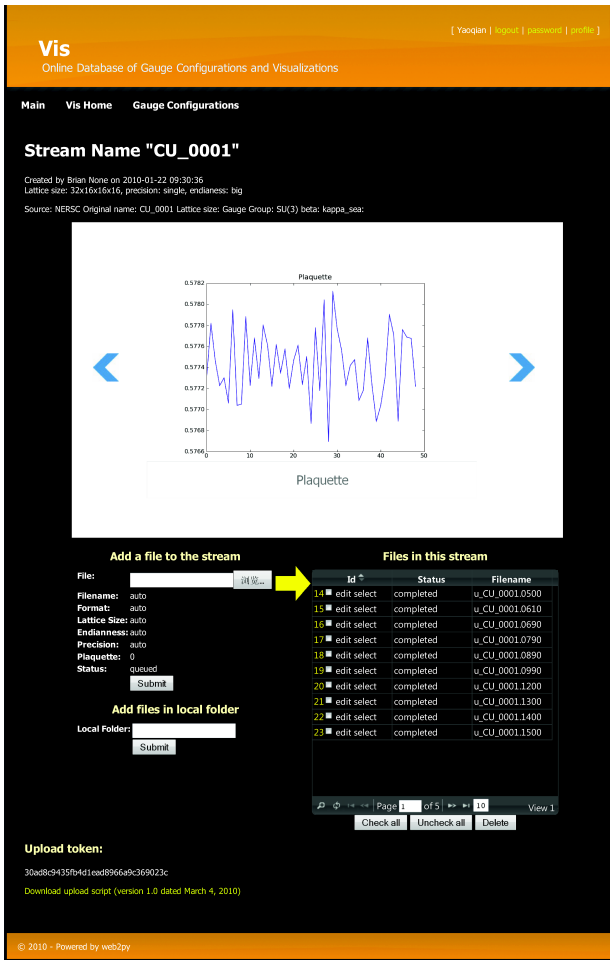


Figure 2.1: Screenshot of the stream view, including the MC history for the average plaquette and a list of files in the stream.

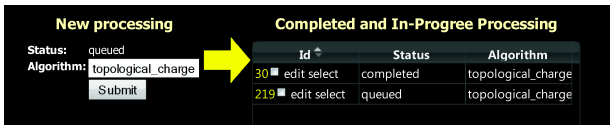


Figure 2.2: Screenshot of the processing view, showing information about the processing of files in a stream, including the status of the computations and the algorithm that was used.

production computations, they do not significantly change the total topological charge. The question, therefore, is whether or not they change the local topological charge density. If they do not, then the computation is biased because the MCMC must get stuck in a topological sector and is not sampling properly. Visualization techniques can be applied in this case, because we have, for example, been able to use them to show that the answer is yes – typical production computations are in fact not stuck in a topological sector.

Our goal is to automate the workflow of physicists working with this data and allow them to:

- store and share gauge configuration for multiple MCMC streams.
- schedule computing jobs for each stream, in particular the computation of the topological charge density.
- visualize the topological charge density (and other derived fields) using isosurfaces and/or volume plots.
- interact with the data using a web interface (rotate the topological charge and change visualization parameters).

2.2 Implementation

Vis, at its core, is a web application for storing collections of datasets and scheduling computations on the files in the sets. A set here is a MCMC stream, and the files in the set are the gauge configurations. Computations can be numerical algorithms and/or visualizations. The computations are submitted via an available Portable Batch System installation and can be parallel jobs.

Users can create an account in the system, login, and perform operations such as creating a new stream, uploading files into the stream, scheduling computations, searching and downloading streams submitted by other users, and viewing the results of computations performed on streams already in the system.

Some computations are scheduled automatically when data is uploaded, because new files need to be explored in order to detect their structure, they must be converted to a standard format, and then analyzed to extract some basic physical parameters that are important for cataloging the file and detecting possible errors.

Individual files can be very large (100M-1GB each) and therefore it may not be practical to upload them via the web interface, which does not support pausing and resuming. To avoid this problem, the system provides an alternate upload mechanism. When a new stream is created, a security token is issued to the user. The user can utilize a provided program to automatically upload every file from a local folder, authenticating via the downloaded token.

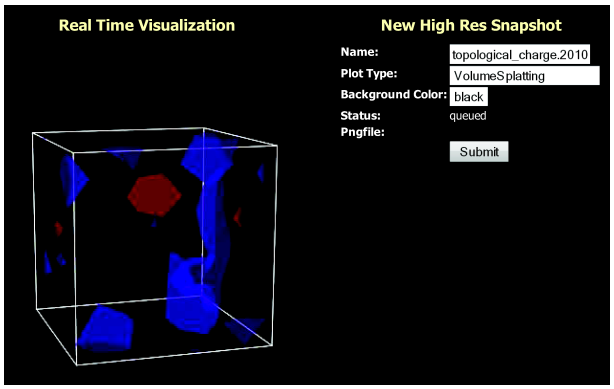


Figure 2.3: Screenshot showing widget that allows limited manipulation of visualized datasets.

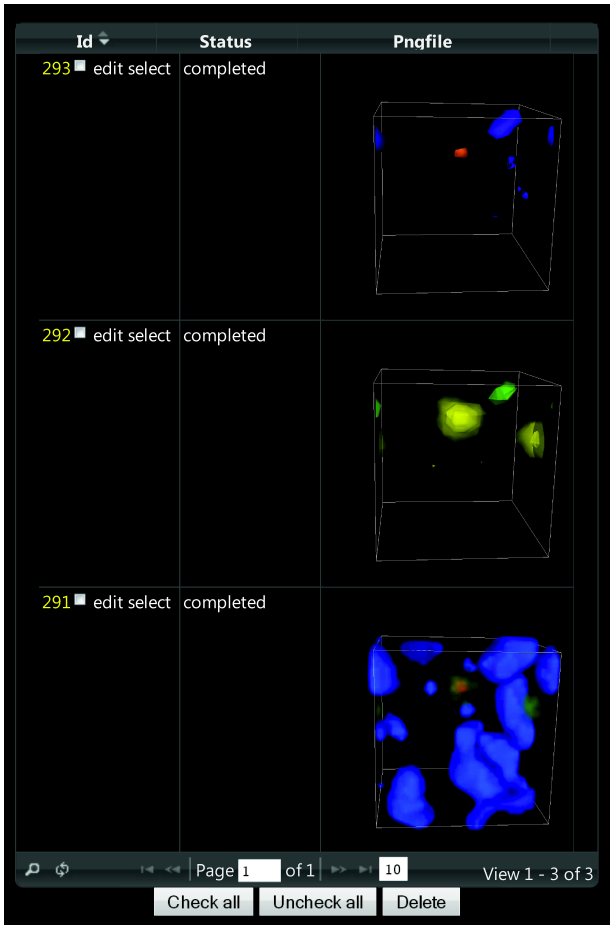


Figure 2.4: Using the browser-based interface, the user can set parameters for the visualization and then submit the job for processing.

The algorithm that computes the topological charge density generates output in VTK format. These files can, to a limited extent, be manipulated via the browser using a JavaScript widget that displays isosurfaces at 60% of the dataset's minimum and maximum values.

The user can choose a visualization angle, specify additional parameters, and then schedule a full visualization job to generate a high-res image.

The web interface was built using web2py [Di Pierro, 2010]. We utilize matplotlib [Hunter et al., 2010] for 2D plotting and VisiIt for 3D visualization of volume plots and iso-surfaces, although we are exploring the possibility of moving to Vish [Benger, 2010] for the latter.

2.3 Summary

At this point Vis is primarily in the prototype stage, because it is hosted on a small PC and lacks the computing resources and bandwidth to transfer and store very large files. However, the program is fully functional and has been used to process some streams of gauge configurations that are made freely available by various groups via the NERSC archive [U.S. DOE, 2010].

Visualization algorithms can help physicists gain new insights into the physics of QCD, and Lattice QCD computations in particular. Our hope is that Vis can lower the barrier of entry, and enable physicists to look more deeply into their data. Vis can be downloaded from: `https://launchpad.net/qcdvis`

Acknowledgments

Project funded by the Department of Energy grant DEFC02-06ER41441.

Bibliography

[Altarelli, 2002] Altarelli, G. (2002). A QCD primer. URL: `http://arxiv.org/abs/hep-ph/0204179v1`.

[Benger, 2010] Benger, W. (2010). Vish website. URL: `http://vish.origo.ethz.ch/`.

[Di Pierro, 2010] Di Pierro, M. (2010). web2py website. URL: `http://web2py.com`.

[Hunter et al., 2010] Hunter, J., Dale, D., & Droettboom, M. (2010). matplotlib website. URL: `http://matplotlib.sourceforge.net/`.

[Massimo Di Pierro, 2006] Massimo Di Pierro (2006). AN ALGORITHMIC APPROACH TO QUANTUM FIELD THEORY. *International Journal of Modern Physics A*, 21, 405–448.

[Novaes, 1999] Novaes, S. F. (1999). Standard model: An Introduction. *Proceedings of the X J. A. Swieca Summer School*. URL: `http://arxiv.org/abs/hep-ph/0001283v1`.

[U.S. DOE, 2010] U.S. DOE (2010). Nersc website. URL: `http://qcd.nersc.gov/`.

Article 3

Increasing Hardware Utilization for Peta-Scale Visualization

Paul A. Navrátil[1], Donald S. Fussell[2], Calvin Lin[2]

[1]Texas Advanced Computing Center
[2]Department of Computer Science

The University of Texas at Austin
Austin, Texas 78712

The dawning of the petascale era of scientific computing has brought with it the challenge to analyze incredible amounts of data. Because it is infeasible to move such data away from the supercomputers that produce them, visualization and data analysis software must be performed on the actual supercomputers themselves. Visualization applications, however, are not currently designed to take full advantage of the computational resources on modern supercomputers nor to operate under their stricter per-core memory limits. To explicitly leverage both process- and thread-level parallelism and to utilize graphics hardware-based computation acceleration, many solutions have been proposed that would require significant reimplementation or replacement of existing software. Unfortunately, it is not clear that such wholesale change is desirable or even possible.

We instead propose an incremental path to migrating visualization software onto petascale machines while achieving high utilization of all computing resources available. Our library-based approach comprises a set of MPI extensions that elevate graphics hardware to first-class entities with respect to data movement and communication, a fine-grained

thread scheduling library that enables dynamic and adaptive task- and data-parallel processing on both CPUs and attached devices such as GPUs, and a library that dynamically blocks incoming data to sizes appropriate for the target machine architecture. This software infrastructure will provide a viable path for enhancing current software to allow improved efficiency on multicore CPU-GPU cluster-based architectures with reasonable programming effort while supporting future hardware innovations.

3.1 Introduction

Scientific visualization has traditionally been a second-class supercomputing problem, a task that could be shipped to a small machine with graphics hardware after the "real" task of performing some large-scale simulation was complete. However, as simulations produce ever larger volumes of data as output, this approach is no longer feasible because there is insufficient bandwidth for shipping terabytes—and soon petabytes—of data off of the supercomputer. Thus, high end visualization must now run on the same HPC hardware that produces the data, and visualization has become an HPC application in its own right.

HPC hardware typically consists of a distributed memory supercomputer whose nodes have shared memory multi-core chips and, increasingly, a set of attached GPUs. Indeed, given the energy and price/performance benefits of GPUs, we are increasingly seeing clusters of GPUs where the GPU is the primary compute engine. Unfortunately, HPC applications typically make poor use of this disparate hardware. Some solutions focus solely on utilizing the GPU [Dolbeau et al., 2007, Strengert et al., 2008]; other solutions attempt to exploit both CPUs and GPUs but are limited to a single node; while still others recognize the need for a hybrid model such as MPI/OpenMP [Lusk & Chan, 2008, Rabenseifner et al., 2009] or MPI/Pthreads [Pfeiffer & Stamatakis, 2010] that distinguishes between the distributed memory and shared memory components of the hardware but ignores GPUs. High-end visualization software is further behind the adoption curve. Most "parallel" visualization applications utilize only CPU-hosted, thread-based parallelism; few exploit process-based parallelism [Childs et al., 2005, Kitware, 2010a]; and no general application broadly applies hybrid design [Vo et al., 2010]. To further complicate matters, visualization is often irregular both in its data structures and its access patterns.

Given this landscape, we believe that peta-scale visualization will only be viable if we are able to significantly increase its utilization of the available hardware. To solve this problem, we propose to design and develop GAMPI, an extension of MPI that gives programmers seamless access to both CPUs and GPUs. GAMPI

provides a hybrid model in which programmers use MPI-like facilities to move data and perform computations across the various distributed memory nodes; within a node programmers can use threads with a default thread scheduler for both CPUs and GPUs, or for more irregular computations, programmers can use an expanded thread interface to obtain improved dynamic scheduling across both CPUs and GPUs. More specifically, this expanded thread interface allows programmers to provide a flexible resource-aware priority scheme, to dynamically dice tasks into smaller pieces to improve load balance, and to use a set of synchronization primitives that have extremely lightweight implementations. Finally, because programmers are typically unaware of the performance benefit of blocking data, we propose a library that dynamically reformats data into blocks that are appropriate for the target machine.

Our approach offers several benefits. First, it is a comprehensive solution that incorporates all available hardware resources. Second, because it builds upon MPI rather than CUDA or OpenCL, GAMPI offers an incremental development path for the large base of existing MPI applications. Third, this incremental development path allows programmers to use existing tools and programming environments, rather than having to learn a new language and its associated tool chain.

The remainder of this position paper describes in more detail the challenges facing high-end visualization (Section 2), the need for supporting dynamic variable-grained thread management (Section 3), and contrasts our approach with prior work (Section 4) before briefly concluding.

3.2 Visualization Challenges

There are several challenges to performing visualization on supercomputers. First, many visualization and analysis packages are sequential programs that cannot exploit the massive parallelism available in supercomputing clusters. Second, visualization operations and their internal data formats are typically not designed for the tight memory limits of HPC production systems; thus, a single node cannot hold both the data and the additional overhead from the visualization application. Third, the packages that do run on clusters do not parallelize their visualization pipeline: To obtain sufficient RAM to support in-core processing of the largest data sets, supercomputer nodes must be allocated with a single process per node, leaving many cores idle that could be included in the visualization and analysis work.

Available memory per-core might be the largest limiting factor for hardware utilization. Due to the per-core memory limits on current machines, HPC codes typically use algorithms and data structures highly-optimized to reduce memory consumption. Unfortunately, visualization applications have yet to broadly adopt

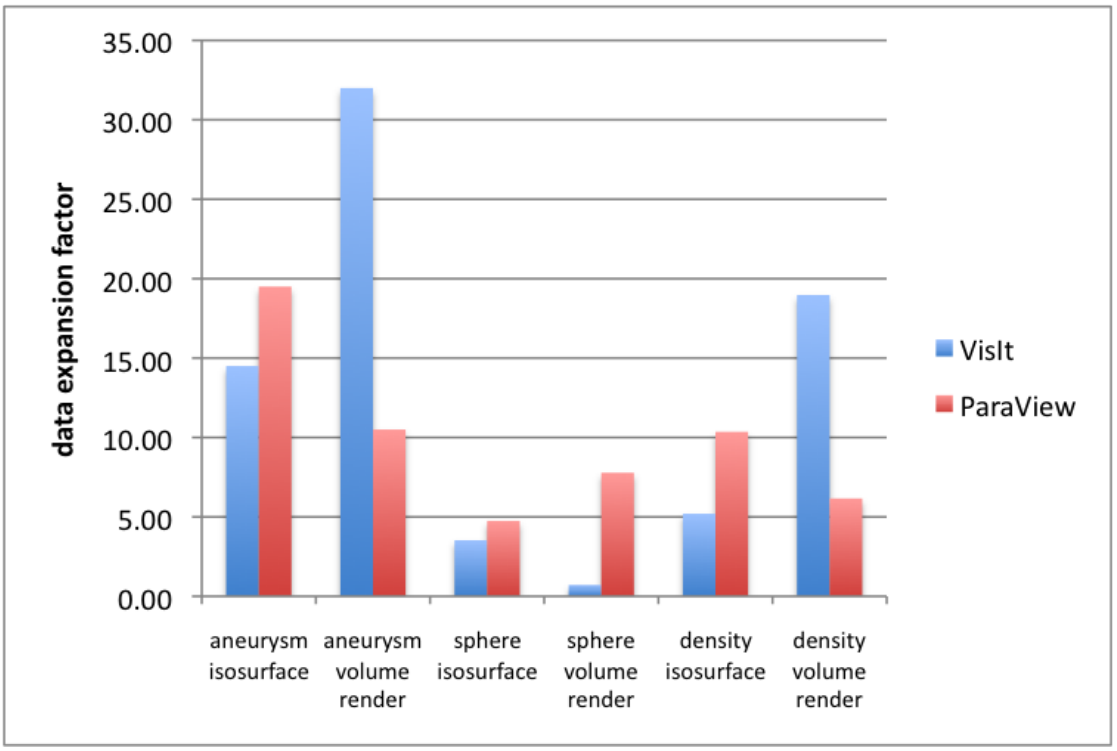


Figure 3.1: This figure shows the memory overhead incurred by isosurfacing and volume rendering under VisIt and ParaView, two popular VTK-based visualization applications. Each dataset was visualized using 128 processes on the TACC Longhorn cluster.

this practice. Figure 3.1 demonstrates the memory overhead of two popular VTK-based applications [Childs et al., 2005, Kitware, 2010a] when performing isosurfacing and volume visualization several datasets:

Brain Aneurysm — 512^3 shorts, 256 MB (128 blocks $\times$ 2 MB)

Synthetic Sphere — 1024^3 floats, 4096 MB (128 blocks $\times$ 32 MB)

Dark Matter Density — 2048^3 integers

- SILO format — 33280 MB (512 blocks $\times$ 65 MB)
- VTK format — 44544 MB (512 blocks $\times$ 87 MB)

When running these applications on a supercomputer with limited memory per core, this data expansion requires the use of fewer processes per node and thus leaves processors idle unless the application can exploit thread-based parallelism.

These challenges lead to poor node utilization, particularly for large-scale data. Yet, there are opportunities to improve the utilization with evolutionary changes to the current software infrastructure.

3.2.1 Pipelined Task Parallelism

Current generation VTK-based applications such as ParaView and VisIt provide coarse-grained parallelism based on a streaming model of dataflow through a set of visualization filters. This is a very flexible scheme that allows easy reconfiguration of the system to various visualization tasks and is natural to parallelize at the granularity of the individual filters. Given the basic streaming assumption, fine-grained parallelism is achieved using SIMD approaches. While these are easily adapted to SIMT for GPU implementations, they do not extend well to irregular computations that would more naturally rely on fine-grained multithreading. Extensions like Hyperflow [Vo et al., 2010] and VTKEdge [Kitware, 2010b] give coarse-grained threading, but lack the fine-grained control to maximize data-parallel processing of in-core data.

Our approach supports both *dynamic* and *flexible* thread scheduling, which allows an application to adapt the thread allotment to process additional data or to pipeline tasks as appropriate for the resources available. This automatic flexibility allows a single codebase to adapt to various computing platforms, particularly the node configurations across various supercomputers on which a codebase will be executed.

3.2.2 GPU-Accelerated Parallelism

With the explosion of GPGPU algorithms and the increasing frequency with which GPU devices are installed in large-scale clusters, GPU-accelerated visualization operations have become a viable means to achieve improved performance. As with the CPU case above, however, visualization algorithms will benefit from fine-grained control over the threads running on a graphics device. GPU thread schedulers are optimized for a regular and balanced distribution of work across threads. Thus they are not ideally suited for for irregular visualization algorithms, such as isosurfacing or streamlining, where the workload depends on the classification of the data at each particular block. Further, the load can vary dynamically as the input criteria to the visualization operation change, such as during data exploration. Here too, we expect fine-grained thread control to increase GPU utilization by providing a seamless environment to swap among task kernels within a single über-kernel

3.2.3 Data Redistribution

Data that has been blocked for efficient computation within a simulation does not always match the best blocking for visualization. If the visualization runs on the same hardware, the block size should be reduced to include the visualization application overhead; if it runs on different hardware, the blocking may need to be redone entirely for efficient data use. The blocking should also consider the higher levels of the memory hierarchy, including the CPU caches and any GPU memory. In addition, visualization algorithms keep data in an internal representation that typically is not optimized for the hardware where the application is running.

We propose to dynamically reblock data to fit upper levels of the memory hierarchy to achieve higher memory utilization and therefore improved application performance. This operation is straight-forward for data in regular data, which we can reblock automatically, but it can be significantly more complex for unstructured or multi-resolution data. For these more complex formats, we provide the user a *reblock* function to guide data reallocation. We expect to increase the capability of our automated reblocking functions as our library matures.

3.3 Dynamic Thread Management

Our thread management library is designed to support workloads that contain a mix of task parallelism and data parallelism and that have variable, dynamically changing granularity. Our model supports any shared memory system, and the focus for this project is to improve scheduling both for multicore CPUs and for GPU-style throughput-oriented architectures. Our primary goal for these platforms is to enable irregular computations to achieve levels of parallel efficiency comparable to those achievable by traditional, more regular workloads. Because GPUs and other throughput-oriented machines make heavy use of multithreading to hide memory latency, it is critical that we schedule such workloads to make effective use of all critical hardware resources, even when the demand on different system resources (e.g. processors, buses, memory system) can change dynamically. For example, in a computer game that includes a physics simulation as well as a renderer (among other tasks), the dynamic nature of the physics simulation (e.g., the increasing effect of an explosion) would cause our scheduler to adapt its schedule in response to the physics simulation's increased use of memory bandwidth.

Such adaptive scheduling can quickly become extremely complex, and it becomes quite burdensome when undertaken explicitly by a programmer. Current generation GPUs solve this problem by optimizing for the regular case; thus, they provide simple hardware-based thread schedulers, but these have proven inflexible, particularly when used on irregular tasks for which the hardware was not originally

designed, leading to seriously suboptimal performance on such computations [Aila & Laine, 2009]. Our system provides a relatively simple to use software library that allows the programmer to tune a much more flexible scheduler for performance on a given application while not having to write low-level code to perform the thread management directly themselves.

Our scheme works by identifying three separate mechanisms for thread synchronization. The first is for hierarchical, or parent-child dependences between threads. Our scheduler provides an operation called *dicing* that allows a thread to be cut into smaller pieces and replaced by finer-grained threads, thereby improving load balance. Our task management scheme and the synchronization among the threads is carefully managed using wait-free synchronization methods to provide minimal performance degradation in this case.

We also provide two levels of locks for more general dependences, which are likely to be found at more coarse-grained levels of the system. There are two classes of such locks, short-lived and long-lived, which are distinguished declaratively by the programmer, primarily on the basis of whether or not full-blown mutual exclusion is expected to be required among the threads in question. If it is, then long-lived locks must be used; otherwise short-lived locks can be employed with lower overhead.

Another key feature of our system is the ability to assign dynamic priorities to threads. We have a two-level priority scheme to allow threads from multiple independent processes to be handled concurrently. The high order portion of a thread's priority is that of its parent process; this portion of the priority is simply a number assigned statically at load time. The low order portion is derived from a priority specification provided by the application programmer in the form of a function rather than a single number. The priority function has as parameters availability levels of various critical system resources. These can include available memory bandwidth, storage availability at various levels of a memory hierarchy, communication bandwidths at various points in the system, and of course available computational resources. These resources vary in their availability as computations proceed, and the variables reflect the instantaneous availability of these resources. This priority scheme allows our runtime system to compute a thread priority dynamically whenever a scheduling decision needs to be made, which allows efficiency gains by dynamically tailoring task scheduling priority to resource availability.

Finally, our system allows programmers to tailor the overall scheduling policy for a process by specifying rules that are optimized for that computation. For advanced programmers this allows a great degree of control over scheduling without requiring them to build the entire scheduling infrastructure; at the same time, this system still allows their computation to share resources with other computations as appropriate. For programmers who do not wish to take advantage of this ability,

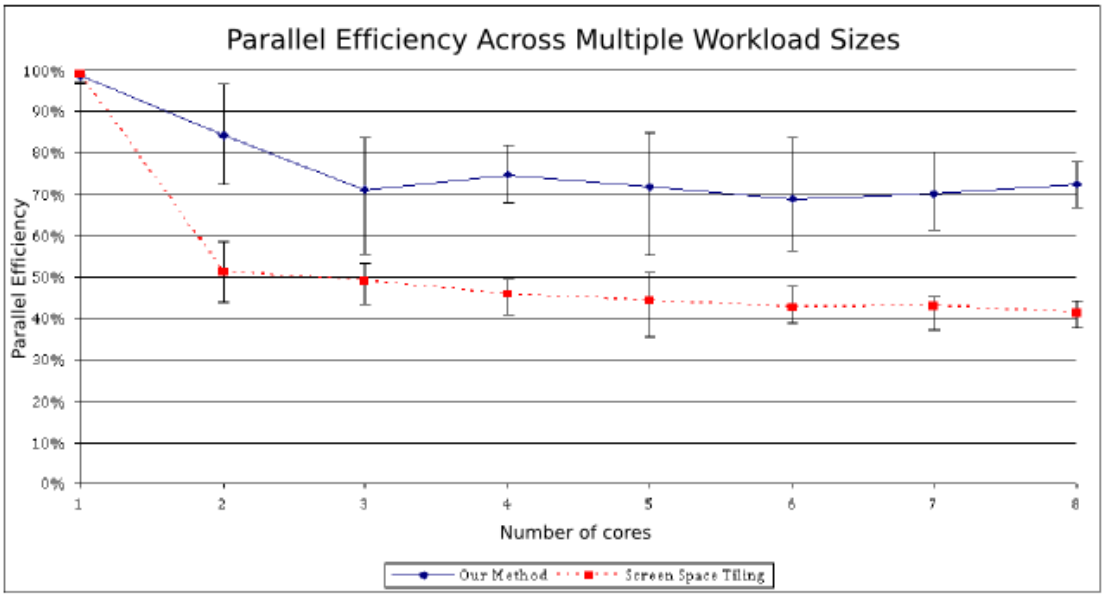


Figure 3.2: Ambient occlusion on an eight core Intel Xeon chip for a range of workloads (4 to 512 spherical samples). For both curves, efficiency is computed using a highly optimized sequential baseline. Our system (top curve) is roughly 75% more efficient than the commonly used Screen Space Tiling approach.

a flexible generic policy is provided as a default.

We have implemented our initial system on a number of multicore CPU platforms in addition to porting it to Intel's internal Larrabee (now Knights Ferry) and to a Sony PlayStation 3. Our initial testing of the system on a multicore CPU has shown that very high levels of parallel efficiency can be achieved with relatively little programming effort. In Figure 3.2, we show efficiency scaling for an irregular ray-tracing style graphics computation (ambient occlusion [Zhukov et al., 1998]) on an 8 core Intel Xeon machine. Against a highly optimized sequential baseline, our efficiency is significantly higher than that of a commonly-used parallel algorithm for such tasks. Because our scheme is significantly more flexible in scheduling work on the machine than the fixed parallelism strategy, we see a high level of efficiency without the detailed programming effort needed to build a scheduler directly as part of the parallel application. We predict additional benefit can be achieved for throughput architectures like the GPU.

We have recently been using our library to support the development of new research code for advanced rendering tasks. Our experience has been that the library greatly facilitates the rapid development of code that shows surprisingly high levels of parallel efficiency, even in its early stages of development. Since our

computations are similar in many ways to a number of scientific applications, we believe that this bodes well for the use of the library as a smooth upgrade path to much higher levels of performance at the node level of a scalable supercomputer.

3.4 Related Work

This section distinguishes our proposed work with prior work. We divide related work into two parts: work that facilitates communication across hybrid cluster environments; and work that performs thread scheduling, both for heterogeneous computing environments and for graphics-hardware devices.

3.4.1 Hybrid Cluster Communication

The growing number of GPU clusters has motivated research into effective data movement and communication across a distributed set of GPUs. Currently, the vendor libraries for graphics-hardware computing lack support for distributed hardware environments. While research has begun in this area, the majority of work adapts GPU kernels to run across a cluster, either by implementing MPI-like communication for the GPU kernel [Fan et al., 2008, Moerschell & Owens, 2006, Strengert et al., 2008, Stuart & Owens, 2009] or by virtualizing the cluster to appear either as a unified system [Barak et al., 2010, Duato et al., 2010] or as a global partitioned address space [Zheng et al., 2010]. In contrast, cudaMPI [Lawlor, 2009], uses a host-based MPI-like syntax to enable data movement between distributed graphics devices, though the library only moves data between GPUs and supports only one graphics device per host. Our approach also uses a host-based MPI-like syntax to facilitate migration of CPU-only codes to include graphics-hardware acceleration, though our approach permits data movement between a host and non-local devices and supports clusters with multiple devices per host.

3.4.2 Irregular Hybrid Thread Scheduling

Hybrid parallelism, combining both process- and thread-level parallelism across a distributed cluster, has been a popular research area in supercomputing for over a decade, though the majority of published work targets regular computation (for example, StarPU [Augonnet et al., 2009]). Here, we discuss work on multithreading for irregular computation, either on the CPU or GPU.

Thread scheduling for irregular programs has been studied in some depth (for recent examples see e.g. [Kulkarni et al., 2008, Mendez-Lojo et al., 2010]), but we are aware of only one work specifically targeting multithreading in visualization

applications: HyperFlow [Vo et al., 2010] seeks to include coarse-grained task and data parallelism to the VTK framework [Kitware, 2010b] and VTK-based applications. This library specifically targets data streaming operations and does not enable the fine-grained, adaptive parallelism of our approach.

Graphics hardware typically schedules threads in a round-robin fashion appropriate for graphics workloads. Round-robin scheduling is not ideal for irregular workloads, and there have been several efforts to create a more flexible scheduling environment within the limits of both the hardware itself and the exposed hardware interface. One strategy merges kernels at compile-time to allow dynamic scheduling of the individual kernels at runtime [Guevara et al., 2009, Tzeng et al., 2010]. Another strategy builds work hierarchies to balance workloads across GPU processors [Lauterbach et al., 2009, Luo et al., 2010]. A third approach [Gregg et al., 2010] builds host and device versions of targeted functions and dynamically selects between them according to system state and profiling data.

3.5 Conclusion

We propose a library-based approach that supports the evolutionary upgrade of existing HPC visualization tools to peta scale by allowing more efficient use of nodes comprising multicore CPU-GPU hybrids. Our aim is to reduce programming effort in such upgrades by giving programmers a common API that allows them to manage fine-grained thread level parallelism on both the CPU and the GPU of such nodes. Our system handles resource-aware thread scheduling, dynamic thread granularity, reblocking of data, and resource sharing among multiple multithreaded processes by providing library extensions of the MPI library on which current large scale visualization systems are implemented. Our experiences with the use of our thread and resource management libraries on single shared-memory multicore systems suggest that we can achieve parallel efficiencies comparable to those of the best hand-tuned code with much lower programming effort. We expect to achieve similar efficiencies by incorporating our approach into scalable distributed-memory visualization applications.

Bibliography

[Aila & Laine, 2009] Aila, T. & Laine, S. (2009). Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High Performance Graphics* (pp. 145–149).

[Augonnet et al., 2009] Augonnet, C., Thibault, S., Namyst, R., & Wacrenier, P.-A. (2009). StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of Euro-Par*, volume 5704 (pp. 863–874).

[Barak et al., 2010] Barak, A., Ben-Nun, T., Levy, E., & Shiloh, A. (2010). A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. In *Proceedings of PPAAC*.

[Childs et al., 2005] Childs, H., Brugger, E. S., Bonnell, K., Meredith, J. S., Miller, M., Whitlock, B. J., & Max, N. (2005). A Contract-Based System for Large Data Visualization. In *Proceedings of IEEE Visualization* (pp. 190–198).

[Dolbeau et al., 2007] Dolbeau, R., Bihan, S., & Bodin, F. (2007). HMPP: A Hybrid Multi-Core Parallel Programming Environment. In *Proceedings of ACM Workshop on GPGPU*.

[Duato et al., 2010] Duato, J., Peña, A. J., Silla, F., Mayo, R., & Quintana, E. S. (2010). rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters. In *Proceedings of OPTIM* (pp. 224–231).

[Fan et al., 2008] Fan, Z., Qiu, F., & Kaufman, A. E. (2008). Zippy: A framework for computation and visualization on a gpu cluster. *Computer Graphics Forum*, 27(2), 341–350.

[Gregg et al., 2010] Gregg, C., Brantley, J., & Hazelwood, K. (2010). Contention-Aware Scheduling of Parallel Code for Heterogeneous Systems. In *Second USENIX Workshop on Hot Topics in Parallelism*.

[Guevara et al., 2009] Guevara, M., Gregg, C., Hazelwood, K., & Skadron, K. (2009). Enabling Task Parallelism in the CUDA Scheduler. In *Proceedings of PMEA* (pp. 69–76).

[Kitware, 2010a] Kitware (2010a). ParaView: www.paraview.org.

[Kitware, 2010b] Kitware (2010b). The Visualization Toolkit: www.vtk.org.

[Kulkarni et al., 2008] Kulkarni, M., Pingali, K., Carribault, P., Ramanarayanan, G., Walter, B., Bala, K., & Chew, L. P. (2008). Scheduling Strategies for Optimistic Parallel Execution of Irregular Programs. In *Proceedings of SPAA*.

[Lauterbach et al., 2009] Lauterbach, C., Mo, Q., & Manocha, D. (2009). *Work Distribution Methods on GPUs*. Technical Report TR09-016, University of North Carolina.

[Lawlor, 2009] Lawlor, O. S. (2009). Message Passing for GPGPU Clusters: cudaMPI. In *Workshop on Parallel Programming on Accelerator Clusters*.

[Luo et al., 2010] Luo, L., Wong, M., & Hwu, W.-M. (2010). An Effective GPU Implementation of Breadth-First Search. In *Proceedings of the Design Automation Conference* (pp. 52–55).

[Lusk & Chan, 2008] Lusk, E. & Chan, A. (2008). Early Experiments with the OpenMP/MPI Hybrid Programming Model. In *Proceedings of IWOMP* (pp. 36–47).

[Mendez-Lojo et al., 2010] Mendez-Lojo, M., Nguyen, D., Prountzos, D., Sui, X., Hassan, M. A., Kulkarni, M., Burtscher, M., & Pingali, K. (2010). Structure-Driven Optimizations for Amorphous Data-Parallel Programs. In *Principles and Practices of Parallel Programming*.

[Moerschell & Owens, 2006] Moerschell, A. & Owens, J. D. (2006). Distributed texture memory in a multi-gpu enviroment. In M. Olano & P. Slusallek (Eds.), *Graphics Hardware 2006*.

[Pfeiffer & Stamatakis, 2010] Pfeiffer, W. & Stamatakis, A. (2010). Hybrid MPI/Pthreads Parallelization of the RAxML Phylogenetics Code. In *Proceedings of HiCOMB* (pp. 1–8).

[Rabenseifner et al., 2009] Rabenseifner, R., Hager, G., & Jost, G. (2009). Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of Euromicro PDP* (pp. 427–436).

[Strengert et al., 2008] Strengert, M., Müller, C., Dachsbacher, C., & Ertl, T. (2008). CUDASA: Compute unified device and systems architecture. In *Proceedings of EGPGV*.

[Stuart & Owens, 2009] Stuart, J. & Owens, J. D. (2009). Message passing on data-parallel architectures. In *Proceedings of IPDPS*.

[Tzeng et al., 2010] Tzeng, S., Patney, A., & Owens, J. D. (2010). Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of High Performance Graphics*.

[Vo et al., 2010] Vo, H. T., Osmari, D. K., Summa, B., Comba, J. L. D., Pascucci, V., & Silva, C. T. (2010). Streaming-Enabled Parallel Dataflow Architecture for Multicore Systems. In *Proceedings of IEEE EuroVis*, volume 29.

[Zheng et al., 2010] Zheng, Y., Iancu, C., Hargrove, P., Min, S.-J., & Yelick, K. (2010). Extending Unified Parallel C for GPU Computing. In *SIAM Conference on Parallel Processing for Scientific Computing*.

[Zhukov et al., 1998] Zhukov, S., Iones, A., & Kronin, G. (1998). An ambient light illumination model. In *Rendering Techniques* (pp. 45–56).

Article 4

Portable Direct Manipulation Specification for Scientific Visualization in Virtual Environments

Irene Tedjo-Palczynski[1], Bernd Hentschel[1], Torsten Kuhlen[1]
[1] Virtual Reality Group at RWTH Aachen University, Germany
{tedjo|hentschel|kuhlen}@vr.rwth-aachen.de

Direct manipulation is an important component of exploratory scientific visualization because it provides a suitable user interface for spatial inputs. However, its realization is challenging because of the diversity of input devices and the lack of reuseability. We propose an abstraction of direct manipulation by decoupling the low-level interaction groundwork from high-level declarations of interaction behaviors. Then, we describe how recurring interaction patterns in scientific visualizations can be distilled into 3D widgets, which can be operated as well in virtual environments as in desktop setups. Finally, we show how the proposed concept improves the implementation process of exploratory scientific visualization.

4.1 Introduction

Exploratory visualization complements powerful visualization algorithms in the analysis of scientific data in which not every aspects of the data is known in detail. An explorative visualization method comprises three main components: (1) a real-time *visualization algorithm*, (2) a user interface to interactively define the input

parameters for the visualization algorithm – the *interaction interface*, (3) a link between the algorithm and the user interface, defining the mechanism to adjust the visualization to the current given parameters – the *mediator*.

Earlier work on the *interaction interface* proposed 3D visualization widgets which provide a natural control interface in the visualization environment. Although this work led to promising results, the improvement of this kind of interactions is troubled by the high complexity of realizing new interaction ideas. High requirements on the performance, numerous possibilities of setup for data flow from interaction devices to assigned actions and variation of the devices are some of the obstructing factors. Thus, reusable parts of existing interaction techniques are valuable because they provide the building blocks for more sophisticated techniques. The goal of this work is not only to increase the reusability of recurring interaction techniques, but also to assure the portability across a large variety of available input devices, e.g. desktop-based mouse or 6-DOF pointer.

Our contributions comprise the following steps: (1) We introduce an *interaction vocabulary* to decouple the declaration of interaction techniques from specific input devices. (2) We show how an *interaction interface* is constructed from reusable components which are distilled from recurring interaction patterns for scientific visualization. (3) We describe example use cases which demonstrates the application of the proposed construction, performed by diverse participants throughout the development cycles of interactive scientific visualization.

After reviewing related work in the next section, an overview of the proposed approach is described in Section 4.3. The proposed solution for a device-independent declaration of interaction is then described in Section 4.4. Based on this, the construction of interaction in scientific visualization will be specified in Section 4.5. Section 4.6 shows the results achieved up to now by describing some use cases. The last section discusses the benefits and limitations of the proposed concepts and provides pointers for future work.

4.2 Related Work

In [Bowman et al., 2004] 3D direct manipulation techniques are described as a class of interaction metaphors by which the user can select and directly manipulate virtual objects with her hands. Some advantages of applying direct manipulation in scientific visualization are reported in [Bryson, 2005]. Alongside the natural ways of interaction, the near-real-time responsiveness is mentioned as a main advantage of direct manipulation interfaces.

In most cases the objects occurring in scientific visualization are representing abstract entities by mapping them to geometric primitives. *Visualization widgets* have emerged as an appropriate tool to allow direct manipulation of visualization

objects. Bowman et al. [Bowman et al., 2004] defined a 3D widget as a specialized object which is artificially added to the environment, providing a selectable instance that can be used to control the environment.

One of the first realizations of direct manipulation for scientific visualization is the *Virtual Windtunnel* [Bryson & Levit, 1991]. They introduced a tool called *bubbler*, which allows the user to place seed points for the computation of *streaklines* directly into the "virtual flow". Extending this idea, Herndon et al. proposed the *Probe*, *Rake* and *Hedgehog* widgets [Herndon & Meyer, 1994].

Recognizing that the lack of reusability causes the underutilization of 3D interaction, Conner et al. described a framework for designing, implementing and using 3D widgets [Conner et al., 1992], in which two behavioral aspects of widgets are distinguished: *dependencies* and *controllers*. In [Ray & Bowman, 2007] an approach for reusable 3D interaction techniques by inserting a software layer between the application code and the software toolkit is proposed. An idea of generalizing input devices – defining the *virtual devices* – is introduced in [Foley & Wallace, 1974] . Another approach focuses more on the semantic of the interaction, defining an *interaction language* [Kamran & Feldman, 1983]. The design of the proposed *interaction vocabulary* is based on these ideas.

[Kok & van Liere, 2007] describes an approach that utilizes the widget implementation of the *Visualization Toolkit (VTK)*.We decided not to build our interaction framework around VTK's 3D widgets. First, because they are not designed to be operated by 3D interaction devices. Thus, although it is possible to extend VTK such that it can be operated with various kind of input devices, this approach does not scale well as the complexity of the 3D interaction grows, e.g. due to the usage of more sophisticated 3D selection methods to compensate device's jittering. Second, because VTK is less suitable for performance critical visualization algorithms, and in particular, time-varying datasets. In such cases, special data structures and rendering algorithms are required, e.g. to ensure appropriate time-management and efficient data processing.

4.3 Overview

Generally, a visualization application is built on a *VR toolkit* and a *visualization toolkit*. The term *VR toolkit* does not imply that the application has to run on virtual reality hardware, but rather points to the core function of VR toolkits: to encapsulate the handling of different types of display environments and drivers for various input devices. The *visualization toolkit*, e.g. VTK, provides basic functionality required to generate visual primitives from the input data.

The flow of user input data from the VR toolkit through the application to a visualization algorithm is depicted in figure 4.1. Although a VR toolkit typically

takes over the task of configuring input devices, it does not imply that the visualization widgets automatically become independent from the setup of the input devices. A naive approach would introduce unnecessary device dependencies at the application level, impeding the reuse of the interaction implementation. For instance, the signals from 2D mouse and 6-DOF pointer would have to be distinguished in the application code. Consequently, one single interaction behavior requires more than one implementations, which is hard to maintain and error-prone. For this problem, we propose a high-level interaction definition (the red arrow in figure 4.1) which is described in section 4.4.

The second part of the proposed construction is built on top of the high-level interaction definition. This ensures code-reuse at the levels of general-purpose 3D interaction, of the widget-construction and of the recurring interactive visualization methods. The blue arrow in figure 4.1 illustrates the location of our solution which is described in section 4.5. Overall, the resulting reusable elements can be ordered as a software layer between the application and the software toolkits (the grey box in figure 4.1).

4.4 High-level Interaction Definition

As well in [Foley & Wallace, 1974] as in [Kamran & Feldman, 1983] an abstraction of interaction tasks is proposed in order to encapsulate the development of interaction techniques from specific input devices. In this abstraction, a set of fundamental interaction tasks is defined, each of which can be referred by a meaningful name that describes its behavior.

This concept is comparable to the concept of low-level and high-level programming paradigms. While the low-level programming languages, e.g. assembler, operate on hardware-specific abstraction level, high-level programming languages encapsulate the hardware details. Furthermore, the high-level languages comprise elements that are easier to understand than the low-level ones, providing a more

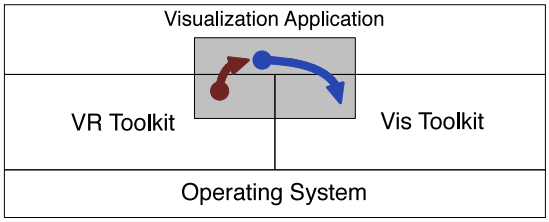


Figure 4.1: Components of the proposed construction: high-level interaction definition (the red arrow, section 4.4), elements of widget-based interaction (the blue arrow, section 4.5), reusable elements for visualization applications (the grey box).

convenient interface to define complex behaviors.

We convey the concept of programming languages to the 3D interaction techniques by characterizing it in the context of interaction development. A low-level definition of interaction behaviors has little abstraction from device's instruction set, e.g. "button 1 is pressed", and thus, has strong dependencies to the configured input device. On the contrary, a high-level definition hides the device-specific operation, and therefore, establishes the portability across various input devices. Furthermore, it intensifies the use of elements from natural language, thus, improving the human readability of the definition, e.g. "select button" instead of "button 1". In return, a high-level definition stimulates the development of user-oriented behavior because the programmable elements are more human-oriented than those of the low-level interaction definition.

Buiding a high-level definition on top of an appropriate low-level definition is a common practice in the context of programming languages. In this regard, a high-level interaction definition should be built on a low-level interaction interface, which is provided by VR toolkits. Furthermore, as the specifications of high-level programming languages comprising the language's basic elements and approriate defined syntax, a high-level interaction definition interface also have to be specified by a set of basic high-level interaction elements which are to be combined according to certain rules in order to build 3D interaction techniques.

The Interaction Vocabulary formalizes the input data flow from a VR toolkit to a visualization application (see figure 4.2). This formalization is build on the basic concept of data flow network, which is successfully used in many VR systems, e.g. [de Haan & Post, 2008]. While the vocabulary is defined by visualization applications (red colored boxes in figure 4.2), an *adapter* (grey colored boxes) is required to configure the data flow from a specific VR toolkit. Hence, the defined vocabulary encapsulates the dependencies of a visualization application from a specific VR toolkit, such that exchanging the underlying VR toolkit would only require the re-routing of input data flow from the adapter to the specified vocabulary.

The vocabulary is defined by a set of *input ports*, each of which must hold a *type* and must be assigned a *name*. The type of a port describes the kind of input information that is required by the application of this port, while its name serves as an identifier that indicates its role in user interaction. Thus, the type of a port can be considered as its "syntax" and the name as its "semantic".

The set of typed and named ports is specified by a visualization application as its interaction groundwork. Based on required basic input information type which is relevant for controlling visualization parameters with widgets, we propose following port types: *boolean* (e.g. the state of input device's button), *integer number* (e.g. a selection from a discrete set), *real-valued number*, *6-DOF* (a 3D position

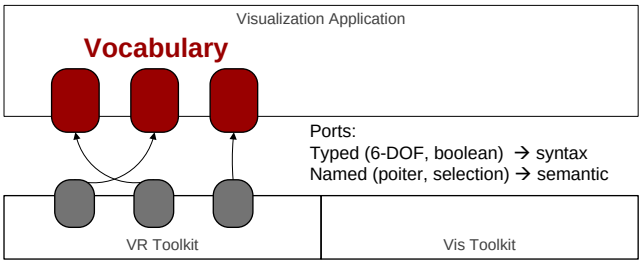


Figure 4.2: The *interaction vocabulary* defines typed and named input ports (red colored boxes). A VR toolkit-specific adapter (grey colored boxes) is required.

and a 3D orientation). It should be noted that these port types rather describe recurring types of 3D interaction components than categorize the types of signals produced by various input devices. Thus, a port of type 6-DOF can be fed by input signals from 2D mouse, analog sticks or a 6-DOF device. Furthermore, the signals from an analog stick, for instance, can alternatively be mapped to control a linear continuous interval which results in a port of type real-valued number.

The following input ports frequently occur in widget-based interactions: (1) A *pointer* (6-DOF) comprises the 3D position and direction of a pointing device. (2) A *selection* (boolean) transmits a button press action expressing a selection command of currently active object. (3) A *viewer* (6-DOF) provides an interface to transmit the current viewer position and direction, which typically comes from a head tracking device in a virtual environments setup. Its output can be used for methods that depend on the viewer's position and orientation.

We implemented this idea by using XML setup files which describe a directed acyclic graph made up of nodes and edges. A node may describe (1) an input data source, which is implemented by the device driver, (2) a transitional instance, e.g. to transform or filter the data values, or (3) the data sink, i.e. the interaction part of the target visualization application. An edge in the graph defines the data flow between two existing nodes by means of a directed connection.

The following simplified code snippet exemplifies such an XML-based setup for a 2D mouse as a 6-DOF pointer. First, we defined three nodes: (1) the input data source "mouse" which routes the signal from the device driver, (2) a node "3d mouse" which converts a 2D position to a 3D position and 3D direction according to a certain algorithm, and (3) the data sink "application".

```
<node name="mouse" type="DriverSensor">
       <param name="driver" value="MOUSE"/>
</node>

<node name="3d mouse" type="3DMouseTransform">
       <param name="displaysystem" value="MAIN"/>
       <param name="viewport" value="MAIN_VIEWPORT"/>
</node>

<node name="application" type="application_context"/>
```

Then, we define the directed connection between above defined nodes. In the code snippet below we route the 2D signal from the 2D mouse driver to the transformer node, then use the generated 6-DOF information(i.e. orientation and position) as an input for the application. The last line defines the mouse's left-button as the "select"-button in the target application.

```
<edge fromnode="mouse" tonode="3dmouse"
      fromport="X_POS" toport="x_pos"/>
<edge fromnode="mouse" tonode="3dmouse"
      fromport="Y_POS" toport="y_pos"/>

<edge fromnode="3dmouse" tonode="application"
      fromport="orientation" toport="pointer_ori"/>
<edge fromnode="3dmouse" tonode="application"
      fromport="position" toport="pointer_pos"/>

<edge fromnode="mouse" tonode="application"
      fromport="LEFT_BUTTON" toport="select"/>
```

A single application may be configured by several setup files, each of which may describe an interaction aspect, such as pointer, selection, etc. Thus, in practice, changing a device setup can be achieved by simply loading different XML files. Our implementation is based on the ViSTA Virtual Reality Toolkit which is avaiable at `http://sourceforge.net/projects/vistavrtoolkit`.

4.5 Widget-based Interaction

Based on [Conner et al., 1992] and [Bowman et al., 2004] we define a widget as an object which is artificially added into the environment to allow interactions directly inside the environment. Hence, the components of a widget are: (1) a *geometric representation* of the widget state, e.g. its position and shape, (2) a *definition of the widget behavior* which specifies the mapping of incoming information from the input ports (e.g. triggered button) to *assigned actions*.

Most existing implementations of visualization widgets do not clearly structure the assigned actions. For instance, a widget controlling a parameter set for a visualization algorithm includes in its behavior definition the following actions: the update of its state, the adjustment of its geometric representation, the computation of the visualization algorithm and also the rendering of the resulting depiction. Hence, reuse of individual components is compromised. The recurring widget-based interaction behaviors would have to be re-implemented for each new combination with visualization algorithms.

We propose to explicitly restrict the influence of a widget's behavior so that it can only directly manipulate its own state. For example, the behavior definition of a box widget that facilitates a selection of a volume-of-interest contains only the resize of the widget's box geometry and not the resize of the selected subvolume. The following descriptions explain the process of combining the proposed widget construction with visualization algorithms.

Based on our observation, we distinguish three roles in the development of widget-based visualizations (see figure figure 4.3): (1) An *interaction developer* designs and implements 3D interaction techniques, including widgets. (2) A *visualization developer* designs and improves interactive visualization algorithms, and combines the algorithm with existing widgets. (3) An *application developer* builds in existing combinations of widget and visualization algorithm – *widget aggregates*

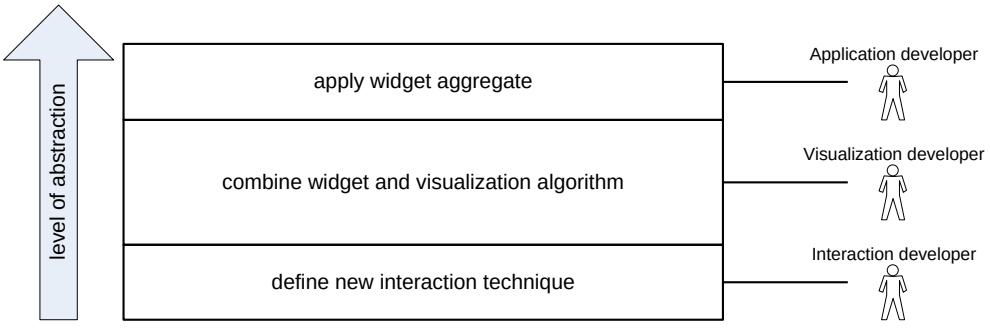


Figure 4.3: The development process of a widget-based interactive visualization.

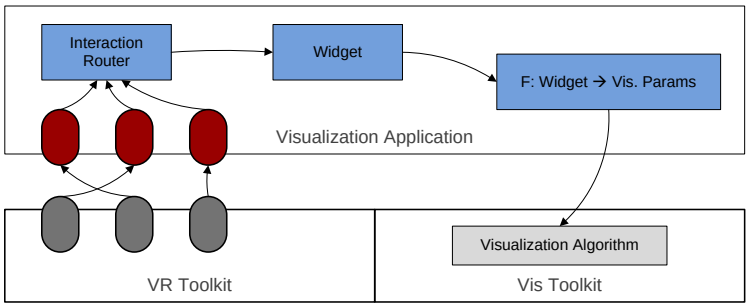


Figure 4.4: Components of widget-based interactive visualization

– into the visualization applications.

The above description of development steps reveals three logical modules of widget-based interactive visualization (see figure 4.4): (1) a widget module, (2) a real-time visualization algorithm, (3) a mapping module which maps the widget output to the input parameters for the visualization algorithm. Furthermore, an *interaction router* is defined to preprocess input data transmitted by the *interaction vocabulary* such that the data is routed to the appropriate widget.

The widget module is structured based on the Model-View-Controller pattern for graphical user interfaces. The *widget model* provides the unique definition of the widget's geometry and other internal states, e.g. the width, height and depth of a box widget. The *widget view* defines the representation of a widget according to its state. The *observer* design pattern ensures that the view is immediately adjusted as soon as a model modification occurs. The *widget controller* is the part of a widget that receives input data from the *interaction router* and converts the input to assigments that modify the model. To determine the possible interfaces, e.g. move or resize widget, a set of selection targets is defined as part of the controller.

To map the widget-based interaction to the control of visualization parameters, a link between the widget and the appropriate visualization to be controlled is defined by using the *mediator* behavioral pattern (see the component "$F : Widget \rightarrow Vis.param$" in figure 4.4).

4.6 Use Cases

We demonstrates the usage of the proposed widget construction with two use cases in which the same plane widget can be reused and combined with two different visualization algorithms (see figure 4.5).

A direct manipulation for volume exploration is often realized by providing an interactive volume slicer. For the interaction part, a plane widget offers a suitable

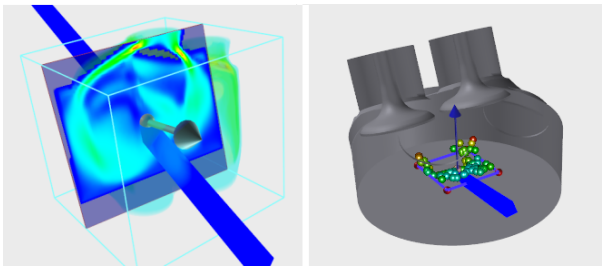


Figure 4.5: Plane widget as interactive slicer (left) and particle-seeder(right).

user interface to position the slicing plane and define its normal vector. The visualization algorithm that generates the slice is not the focus of this work, but its non-trivial implementation should be combinable with other user interfaces. Thus, taking out the dependencies between a visualization algorithm and its controlling user interface by using a mediator profits both the reusability of the interaction widget and the visualization algorithm. In this use case, the mediator transforms the plane model's attributes to the appropriate input parameter for the volume slicing algorithm.

The same plane widget as described in the previous use case can also be applied to establish a 3D interaction to position a quad-formed particle seeder. The manipulation of the quad's normal remains exactly equal as described above. The mediator between both components is then defined to translate the quad area to the area where particle seeds are released into the flow. With the mediator encapsulating the dependencies between the interaction and visualization algorithm to compute particle traces, not only the existing plane widget can be reused, but also the particle tracer can be combined with other seeders.

4.7 Discussion

We proposed an *interaction vocabulary* to allow high-level interaction definition, which encapsulates device-specific operations. The *interaction ports* define the type and the name of available high-level elements, which enable a human-readable definition of interaction behavior. Because the *interaction vocabulary* has a static set of port types, interpreting applied high-level elements to the underlying low-level elements can be accomplished by lightweight operations. Thus, it does not introduce any disturbance to the interaction performance.

To establish reusable components in the development of interactive visualizations, we proposed a construction that allows independent development of widget-based interaction techniques and visualization methods. The link between both elements is then established by a *mediator*.

Although the proposed construction theoretically allows other port types apart from 6-DOF and boolean in the *interaction vocabulary*, we still have no experience in its application. Furthermore, an evaluation of the vocabulary for a wide range of device configurations, e.g. two-handed tracked devices, is regarded as an important future work.

Acknowledgements

This work is partially funded by the German Federal Ministry of Education and Research (BMBF) under grant 03SF0326A.

Bibliography

[Bowman et al., 2004] Bowman, D. A., Kruijff, E., LaViola, J. J., & Poupyrev, I. (2004). *3D User Interfaces: Theory and Practice*. Addison-Wesley.

[Bryson, 2005] Bryson, S. (2005). Direct Manipulation in Virtual Reality. In C. D. Hansen & C. R. Johnson (Eds.), *The Visualization Handbook*. Elsevier.

[Bryson & Levit, 1991] Bryson, S. & Levit, C. (1991). The Virtual Windtunnel: An Evironment for the Exploration of Three-Dimensional Unsteady Flows. In *Proceedings of Visualization '91* (pp. 17–24).

[Conner et al., 1992] Conner, B. D., Snibbe, S. S., Herndon, K. P., an Robrt C. Zeleznil, D. C. R., & van Dam, A. (1992). Three-Dimensional Widgets. In *Proceedings of the Symposium on Interactive 3D Graphics* (pp. 183–188).

[de Haan & Post, 2008] de Haan, G. & Post, F. H. (2008). Flexible Architecture for the Development of Realtime Interactive Behavior. In *Proceedings of the Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (pp. 71–75).

[Foley & Wallace, 1974] Foley, J. D. & Wallace, V. L. (1974). The Art of Natural Graphic Man-Machine Conversation. *IEEE*, 62(4), 462–470.

[Herndon & Meyer, 1994] Herndon, K. P. & Meyer, T. (1994). 3D Widgets for Exploratory Scientific Visualization. In *UIST '94: Proceedings of the ACM Symposium on User Interface Software and Technology* (pp. 69–70).

[Kamran & Feldman, 1983] Kamran, A. & Feldman, M. B. (1983). Graphics Programming Independent of Interaction Techniques and Styles. *SIGGRAPH Computer Graphics*, 17(1), 58–66.

[Kok & van Liere, 2007] Kok, A. J. & van Liere, R. (2007). A Multimodal Virtual Reality Interface For 3D Interaction with VTK. *Knowledge and Information Systems*, 13(2), 197–219.

[Ray & Bowman, 2007] Ray, A. & Bowman, D. A. (2007). Towards a System for Reusable 3D Interaction Techniques. In *Proceedings of Virtual Reality Software and Technology (VRST) 2007* (pp. 187–190).

Article 5

Gaalet - A C++ Expression Template Library for Implementing Geometric Algebra

Florian Seybold
Uwe Wössner

High Performance Computing Center Stuttgart (HLRS),
University of Stuttgart, Germany
seybold@hlrs.de, woessner@hlrs.de

Geometric Algebra is a universal mathematical language, used in many scientific areas. Its graded structure inherits different mathematical concepts. Gaalet (Geometric Algebra ALgorithms Expression Templates) enables the implementation of Geometric Algebra in an elegant and concise way in the programming language C++. By using expression templates techniques and accomplishing grading operations at compile time, algorithms and expressions implemented with Gaalet yield good runtime performance.

5.1 Introduction and Previous Work

5.1.1 Geometric Algebra

Brief Introduction

Geometric Algebra can be defined as a non-degenerate Clifford Algebra over the reals, although this definition may vary in publications from different authors. The

term "Geometric Algebra" was coined by David Hestenes, the "godfather" of Geometric Algebra. He fostered the usage of Geometric Algebra as a common mathematical language, as Geometric Algebra does inherit a lot of different concepts, for example the complex numbers, quaternions and the exterior (Grassmann) algebra, the latter being a foundation of Clifford and Geometric Algebra. David Hestenes [Hestenes & Sobczyk, 1984], [Hestenes, 1999] published some important books on this subject.
Following the definition above, we define a Geometric Algebra

$$\mathcal{G}(p,q) \equiv C\ell_{p,q}(\mathbb{R}) \tag{5.1}$$

with non-degenerate metric tensor signature (p, q) (square of basis vector ${e_{1..p}}^2 = 1$, ${e_{p+1..q}}^2 = -1$), Clifford Algebra $C\ell_{p,q}(\mathbb{R})$ over the real numbers $\mathbb{R}$.
This definition may also be extended to a degenerate metric tensor, resulting in more freedom. On the other side, some theorems David Hestenes formulated for Geometric Algebra hold only for the non-degenerate signature case.

A brief introduction to Geometric Algebra might be given by discussing the geometric product. The geometric product

$$e_i e_j = \begin{cases} e_i \wedge e_j & , i \neq j \\ e_i e_j = \pm 1 & , i = j \end{cases} \tag{5.2}$$

of two basis vectors e_i, e_j either generates another graded element of the algebra or it produces a scalar, depending on the algebra's signature. The graded element generated by the geometric product of two different basis vectors is called a basis 2-blade, or basis bivector, of three different basis vectors a basis 3-blade, or basis trivector, and so on. A scalar is a 0-blade. All these basis blades can be scaled by a coefficient and summed up, which produces a general multivector of the algebra (e.g. $A = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 \wedge e_2 \in \mathcal{G}(2,0)$).
By using specialised multivectors and developing theorems on their operations, expressions can be handled in a coordinate free manner. Important types of multivectors are k-vectors, which contain basis blades of the same grade k.

For a more sophisticated introduction to Geometric Algebra, we would like to refer the reader to literature of David Hestenes [Hestenes & Sobczyk, 1984], [Hestenes, 1999], Chris Doran and Anthony Lasenby [Doran & Lasenby, 2003], as well as Leo Dorst et al. [Dorst et al., 2007].

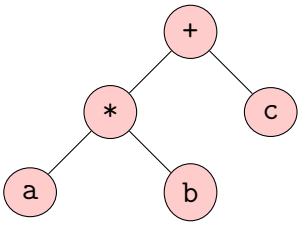
Implementation

It is a common notion that algorithms developed or described in Geometric Algebra take an elegant and compact form, compared to conventional algebras. On the

other side, Geometric Algebra algorithms often tend to yield a worse runtime performance if implemented naively. (For some examples, see [Skala & Hildenbrand, 2009].)
There has been research going on about the implementation of Geometric Algebra expressions and algorithms, mainly concerning the problem to maintain the elegant algorithm description of Geometric Algebra in the implementation, while overcoming common performance issues.
In the next lines, common implementation frameworks for geometric algebra will shortly be described. It will be distinguished between external frameworks, i.e. code generators, which are not embedded into a specific programming language, as well as internal frameworks, i.e. libraries for a specific language.
External frameworks tend to offer high runtime performance by exploiting the circumstance that grading in Geometric Algebra expressions are fixed, thus grading can be handled beforehand and only multivector coefficients are subject to runtime computations.
Gaalop [Hildenbrand et al., 2010] uses an Geometric Algebra specific language (CLUScript) for the description of expressions and algorithms and compiles these descriptions for certain target languages and platforms. It additionally applies heavy simplifications to expressions on the multivector coordinate basis and can examine whole algorithms for optimisations purposes.
Gaigen [Fontijne, 2006] can produce optimised implementations of Geometric Algebra elements and operations for certain target languages. It specialises the implementation of Geometric Algebra for a certain metric and generates an implementation in form of a library the programmer can use.
There are libraries available for specific programming languages, modelling the Clifford algebra or Geometric Algebra. To our knowledge, these libraries deal with the grading of Geometric Algebra elements at runtime, thus producing a certain overhead. In the following a short assortment of C++ libraries are listed.
GluCat [Leopardi, 2007] is a templated library and models Clifford algebra over the reals of arbitrary dimension and arbitrary signature in accordance with its author. It has been originally designed to be used with other generic C++ libraries.
C++ MV [Bell, 2004] models Clifford algebra over the reals of up to 63 dimensions. Its grading implementation is quite sophisticated.

5.1.2 Expression Templates

Expressions Templates is a technique to describe expressions as types in the programming language C++. Templates form the generic part of C++, which enables the programmer to define a C++ class or functions using template arguments. Latter can be types or integer constants, thus the programmer does not know these



(a) Expression tree

```
typedef
addition<
  product<
    a_type,
    b_type>,
  c_type> expr;
```

(b) Expression type in C++

```
a_type a;
b_type b;
c_type c;
auto expr = a*b+c;
```

(c) Definition via operator overloading

Figure 5.1: Expression $a*b+c$: Modelled with expression tree (left), implementation type in C++ (middle) and concise definition of an expression, enabled by operator overloading (right). (Note upcoming standard C++0x keyword `auto`.)

types or constants when defining the class or function. Not until the templated class is used with template arguments to declare an object, or the templated function is called with template arguments.
Figure 5.1 shows an expression tree, modelling an example expression and how the C++ templated class type might look like. It is defined by nesting templated, unary or binary operator classes as template arguments into parent operator classes. These templated operator classes are the nodes of the expression tree, with the special case of an end node, which might be an algebra element, e.g. a real number or a vector, as well as another predefined expression. Using the expression templates technique in conjunction with operator overloading allows for writing expressions in C++ in a concise, domain-specific form.
Expression Templates model the concept of lazy evaluation, thus an expression must not be evaluated when it is defined. This enables the compiler to handle the whole expression at compile time, inlining function calls and avoiding creation of temporary values. Vector-valued expressions can be evaluated on a coordinate basis, without creating temporary vectors. Using expression templates result in potentially faster code compared to a naive implementation, as Todd Veldhuizen [Veldhuizen, 1995] or David Vandevoorde [Vandevoorde & Josuttis, 2003] show. Both developed the expression template technique independently.
Recently Jochen Härdtlein et al. [Härdtlein et al., 2009] published advanced expression templates techniques, which, along others, reduce the complexity of expression templates, both on the implementation side and in the depth of nested template classes, which reduces compile time. This is done by applying the *curiously recurring template pattern* (CRTP) mechanism to the definition of expression tree nodes.

5.1.3 Applications in Visualisation

Important applications of Geometric Algebra algorithms are in Computer Graphics, Physics and Engineering. Especially the domain of visualisation takes advantage of Geometric Algebra, with a lot of potential for future work in this domain. See for example proceedings in [Skala & Hildenbrand, 2009].

As an example for an application in visualisation, we would like to refer to the problem of camera navigation in a visualisation framework. Werner Benger et al. [Benger et al., 2009] describe a Geometric Algebra algorithm used for camera navigation in the VISH [Benger et al., 2007] visualisation framework. The algorithm rotates a camera located at a point defined by position vector $P \in \mathbb{R}^3$, looking at a point defined by position vector $L \in \mathbb{R}^3$, around the view direction vector $t = L - P$ about an angle $\varphi \in \mathbb{R}$. This is simply described by the rotor

$$R_t = e^{-\frac{1}{2}\varphi\left(\frac{t}{|t|}\right)^\star}, \tag{5.3}$$

denoting the rotation of the camera.
The magnitude $|t|$ is used to normalise the view direction vector t. The star in the expression $(\frac{t}{|t|})^\star$ denotes the dual operator, yielding the plane normal the normalised view direction vector $\frac{t}{|t|}$ in form of a bivector. Multiplying the bivector by the negative half rotation angle and operating the exponential function on it yields the camera rotation about angle φ, denoted by rotor R_t. Benger et al. claim that this formulation is considered to be very simple compared to respective formulations using matrix and linear algebra.
A rotor R in Geometric Algebra is similar to a quaternion and may describe the rotation of a rigid object. For example, in order to rotate vector a using rotor R, the sandwich product RaR^{-1} can be applied.

5.2 The Motivation

5.2.1 Geometric Algebra Implementation

Implementations of Geometric Algebra algorithms created with help of the external frameworks Gaalop and Gaigen have shown to yield good runtime performance. Nevertheless, we think that their programming language independence can have negative impact on the development process of software using Geometric Algebra algorithms, although this opinion might be well countered by others. To make the reasons for this opinion clear, the implementation process of a Geometric Algebra algorithm and its integration into software is shortly discussed, both when using Gaalop and Gaigen.

As input, Gaalop explicitly uses a Geometric Algebra specific language, called CLUScript, for the description of Geometric Algebra algorithms and expressions. Compiling such algorithms or expressions written in CLUScript, Gaalop generates optimised code snippets for supported target languages. These code snippets don't represent the underlying coordinate-free description of a Geometric Algebra algorithm or expression anymore, but come in form of optimised code of a coordinate based evaluation. Furthermore, these code snippets have to be manually integrated into the actual software implementation. Beside the additional manual work of connecting generated code snippets with the application code, debugging of algorithms might get very hard. Note that there is already work going on to overcome these problems in form of a compiler driver [Charrier & Hildenbrand, 2010], similar to the NVIDIA CUDA Compiler Driver.
Gaigen generates specialised libraries for a specific Geometric Algebra in terms of its metric. As far as we know, Gaigen doesn't use operator overloading for C++, so the algorithm implementation lack a concise way of writing. There is still runtime overhead due to grading operations, which can be optimised out by profiling (analysis at runtime), with the cost of additional development work.
The C++ libraries GluCat and MV C++ must in general deal with grading of Geometric Algebra elements at runtime, thus keep a disadvantage concerning runtime performance.

5.3 Our Approach

5.3.1 Overview

Our approach is to combine Geometric Algebra and Expression Templates. It is presumed that the types of the input multivectors to an algorithm is known when implementing the algorithm. Thus grading of Geometric Algebra elements doesn't have to be handled at runtime, but can be done beforehand. The idea is to use the metaprogramming capabilities of C++ templates in conjunction with the Expression Templates technique in order to handle grading at compile time. What remains at runtime are the computations with multivector coefficients, in an efficient way which the expression templates technique provides.

5.3.2 Multivector implementation

A multivector class consists of an array of arbitrary size for storing coefficients of a multivector, and a templated list of integer constants, mapping the coefficients to basis blades, as well as member functions for accessing coefficients. The integer list, mapping coefficients to basis blades, allows for storing only the needed coef-

ficients in a multivector. This is sometimes referred to as compressed multivector. Note that at runtime, only the coefficients are accessed, and the constant map is normally only used for grading operations at compile time.
A bitmap representation for basis blades, as described amongst others by Daniel Fontijne [Fontijne, 2007], is used in order to store them in a templated constant integer list. A bit denotes a specific basis vector, the combination of bits denote the outer products of basis vectors, thus basis blades, for example:

$$001_b \equiv e_1, 010_b \equiv e_2, 100_b \equiv e_3, 101_b \equiv e_1 \wedge e_3 = e_{13}$$

Note the strict order of outer product operands. For example bitmap 101_b represents $e_1 \wedge e_3$, not $e_3 \wedge e_1$. Because the outer product of vectors anticommutes, the latter case can be represented with a sign flip in the coefficient ($a_{13}e_1 \wedge e_3 = -a_{13}e_3 \wedge e_1$).
So a definition of a multivector object in Gaalet, for example an rotor $R = a_0 + a_{12}e_1 \wedge e_2 + a_{23}e_2 \wedge e_3 + a_{31}e_3 \wedge e_1 \in \mathcal{G}(3,0)$, looks like this:

```
typedef gaalet::algebra<gaalet::signature<3,0> > em;
em::mv<0,3,5,6>::type R = {a0, a12, -a31, a23};
```

5.3.3 Operation implementation

Unary and binary operations are implemented like shown by Härdtlein et al. [Härdtlein et al., 2009], with additional routines for return type determination. This means that Gaalet determines the type of the resulting multivector object, that is the basis blades it contains, depending on the multivector operand types. For example the geometric product $ab \in \mathcal{G}(3,0)$ of two vectors $a = \langle a \rangle_1$, $b = \langle b \rangle_1$ results in a multivector containing the basis blades of a rotor: $ab = \langle ab \rangle_0 + \langle ab \rangle_2$. In Gaalet, this would look like:

```
typedef gaalet::algebra<gaalet::signature<3,0> > em;
em::mv<1,2,4>::type a = {1,2,3};
em::mv<1,2,4>::type b = {3,4,5};
std::cout << "ab: " << a*b << std::endl;
```

The output of this example would be:

```
ab: [ 26 -2 -4 -2 ] { 0 3 5 6 }
```

The integers in the curly brackets denote the basis blades contained in the resulting multivector.

5.3.4 Evaluation implementation

An important aspect of expression templates is their lazy evaluation capabilities. This is illustrated by implementing the camera rotation algorithm described in section 5.1.3:

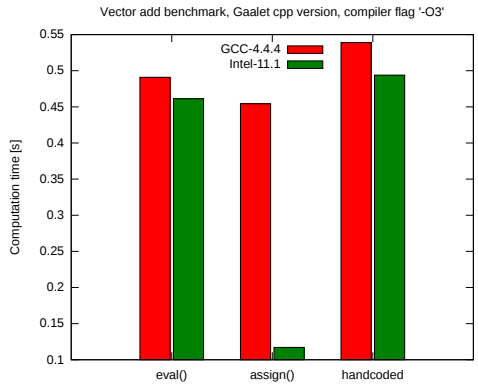
```
//Pseudoscalar for G(3,0):
em::mv<7>::type I = {1.0};
//Position vector declaration of camera and look-at point:
em::mv<1,2,4>::type L, P;
//Camera rotation angle:
double phi;
...
//View direction vector expression defined and evaluated:
em::mv<1,2,4>::type t = L - P;
//Expression defined, not evaluated:
auto S_t = t * !magnitude(t) * I;
//Expression defined, including another expression:
auto R_t = exp(-0.5*phi*S_t);
...
//Rotating vector definition:
em::mv<1,2,4>::type a = {0.0, 0.0, -1.0};
//Expression defined and evaluated into multivector:
auto b = eval(grade<1>(R_t*a*(~R_t)));
```

Without using the `grade<1>()`-operation, the resulting multivector type of `b` would be { `1 2 4 7` }, thus including the pseudoscalar type. Because a sandwich product like `R_t*a*(~R_t)` returns a pseudoscalar coefficient of zero, we can filter it out by using the `grade<1>()`-operation. This results in a multivector type { `1 2 4` }, and a pseudoscalar coefficient is never computed.

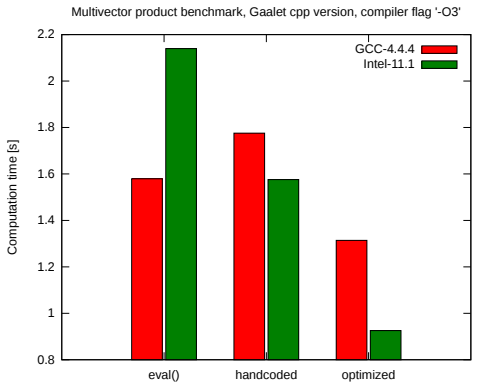
In general, expressions can depend on the multivector the result is stored in, for example in an iteration. Because the coefficients of that multivector might be cross accessed, the result of an evaluated expression must first be stored into a temporary multivector, the content of which has then to be moved or copied into the storing multivector. This procedure is implemented in the multivector member function `operator=()` and the global function `eval()`, latter if used with an existing multivector the result is stored in.
If the global function `eval()` is used for initialising a newly constructed multivector, the compiler should generate code that stores the resulting coefficients directly into the newly constructed multivector. The constructor of a multivector, taking an expression for evaluation as argument, also stores the evaluated coefficients

directly into the newly allocated multivector. Both procedures are used in the example shown above.
The programmer might know that it is safe to store coefficients directly to an existing multivector. He might then use the function `assign()` as well.

5.4 Results



(a) Vector addition benchmark: $c = c + a + b - a - b + a + b - a - b - c \in \mathbb{R}^{12}$, 10^7 times



(b) Multivector product benchmark: $a = ba\widetilde{b}$, $b = \langle b \rangle_0 + \langle b \rangle_2$, $a, b \in \mathcal{G}(3,0)$, 10^8 times

Figure 5.2: Comparing benchmarks of different implementation methods using different compilers: GCC 4.4.4 and Intel C++ Compiler 11.1

While elegance of implemented code might be now and then in the eye of the beholder, we can do performance measures of implementations. In figure 5.2 we show comparing benchmarks of different implementation methods of two artificial problems. Compared are the two different procedures of expression evaluation in Gaalet, either storing resulting coefficients directly to an existing multivector or firstly to a temporary multivector, and a respective hand-coded implementation on a coordinate basis. Two different compilers are used for compiling and optimising code: GNU Compiler Collection 4.4.4 and Intel C++ Compiler 11.1.
The benchmarked problem shown in figure 5.2a is a simple vector addition of the form $c = c + a + b - a - b + a + b - a - b - c$ with $a, b, c \in \mathbb{R}^{12}$, conducted 10^7 times. Of interest here is the optimisation capabilities of the Intel compiler if using the function `assign()` for expression evaluation. In other respects, the Gaalet implementation and the handwritten implementation yield roughly the same performance.
In figure 5.2b, a benchmark of the problem $a = ba\widetilde{b}$ (the tilde denotes a reverse operation) with $a = \langle a \rangle_1 + \langle a \rangle_3$, $b = \langle b \rangle_0 + \langle b \rangle_2$ and $a, b \in \mathcal{G}(3,0)$ is shown,

conducted 10^8 times. Here the function `assign()` is not considered, but two handwritten implementations: One without a symbolic optimisation on a coordinate basis ("handcoded") and one with ("optimised"). One aspect this benchmarks shows, is the dependency of implementation performance, whether handwritten or with Gaalet, on compiler optimisations: The Intel Compiler yields more variance in the performance of different implementations then the GNU Compiler.

5.5 Summary

Geometric Algebra is a universal mathematical language with a graded structure based on the exterior algebra. For the implementation of Geometric Algebra algorithms or expressions, the grading operations produce a certain overhead when done at runtime. In general the grading operations can be accomplished beforehand. Gaalet does the grading operations at compile time by extending the concept of expression templates to grade type determination of a resulting multivector. This is done by metaprogramming using C++ templates, so no external tool is needed besides a C++ compiler (this includes the CUDA device code compiler), resulting in good performance of the Geometric Algebra implementations, while preserving its compact form.

Acknowledgments

While researching for literature on the Internet, we found that Jaap Suter (`http://www.jaapsuter.com`) had the idea of implementing Geometric Algebra with expression templates as well, although no publication or software could be found. One author learnt in private conversation with Robert Valkenburg (Industrial Research Limited, Auckland, NZ), that he also works on a Geometric Algebra implementation using expression templates.

Bibliography

[Bell, 2004] Bell, I. (2004). Multivector programming. URL: http://home.clara.net/iancgbell/maths/geoprg.htm.

[Benger et al., 2009] Benger, W., Hamilton, A., Folk, M., Koziol, Q., Su, S., Schnetter, E., Ritter, M., & Ritter, G. (2009). Using geometric algebra for navigation in riemannian and hard disc space. In *Proceedings of the International Workshop on Computer Graphics, Computer Vision and Mathematics, 2009*: Plzen, Czech Republic, Vaclav Skala - Union Agency. URL: http://gravisma.zcu.cz/GraVisMa-2009.

[Benger et al., 2007] Benger, W., Ritter, G., & Heinzl, R. (2007). The Concepts of VISH. In 4^{th} *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007* (pp. 26–39).: Berlin, Lehmanns Media-LOB.de.

[Charrier & Hildenbrand, 2010] Charrier, P. & Hildenbrand, D. (2010). Gaalop compiler driver. URL: http://gravisma.zcu.cz/GraVisMa-2010.

[Doran & Lasenby, 2003] Doran, C. & Lasenby, A. (2003). *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry*. Cambridge University Press.

[Dorst et al., 2007] Dorst, L., Fontijne, D., & Mann, S. (2007). *Geometric Algebra for Physicists*. Burlington, MA, USA, Morgan Kaufmann Publishers, Elsevier Inc.

[Fontijne, 2006] Fontijne, D. (2006). Gaigen 2: a geometric algebra implementation generator. In *Proceedings of the 5th international conference on Generative programming and component engineering*: ACM. URL: http://staff.science.uva.nl/~fontijne/gaigen2.html.

[Fontijne, 2007] Fontijne, D. (2007). *Efficient Implementation of Geometric Algebra*. PhD thesis, University of Amsterdam.

[Härdtlein et al., 2009] Härdtlein, J., Pflaum, C., Linke, A., & Wolters, C. H. (2009). Advanced expression templates programming. *Computing and Visualization in Science, Vol. 13*.

[Hestenes, 1999] Hestenes, D. (1999). *New foundations for classical mechanics: Fundamental Theories of Physics*. Dordrecht, Kluwer Academic Publishers.

[Hestenes & Sobczyk, 1984] Hestenes, D. & Sobczyk, G. (1984). *Clifford algebra to geometric calculus, a unified language for mathematics and physics*. D. Reidel Publishing Company, Dordrecht, Holland.

[Hildenbrand et al., 2010] Hildenbrand, D., Pitt, J., & Koch, A. (2010). Gaalop - high performance parallel computing based on conformal geometric algebra. In E. Bayro-Corrochano & G. Scheuermann (Eds.), *Geometric Algebra Computing: in Engineering and Computer Science* (pp. 477–494). Springer. URL: `http://www.gaalop.de`.

[Leopardi, 2007] Leopardi, P. (2007). Glucat: Generic library of universal clifford algebra templates. URL: `http://glucat.sourceforge.net`.

[Skala & Hildenbrand, 2009] Skala, V. & Hildenbrand, D., Eds. (2009). *Proceedings of the International Workshop on Computer Graphics, Computer Vision and Mathematics, 2009*. Plzen, Czech Republic, Vaclav Skala - Union Agency. URL: `http://gravisma.zcu.cz/GraVisMa-2009`.

[Vandevoorde & Josuttis, 2003] Vandevoorde, D. & Josuttis, N. M. (2003). *C++ templates: The Complete Guide*. Pearson Education, Boston, MA, USA.

[Veldhuizen, 1995] Veldhuizen, T. (1995). Expression templates. *C++ Report, Vol. 7*.

Article 6

OpenWalnut – An Open-Source Visualization System

Sebastian Eichelbaum[1], Mario Hlawitschka[2], Alexander Wiebel[3],
and Gerik Scheuermann[1]
[1]Abteilung für Bild- und Signalverarbeitung,
Institut für Informatik, Universität Leipzig, Germany
[2]Institute for Data Analysis and Visualization (IDAV), and
Department of Biomedical Imaging, University of California, Davis, USA
[3] Max-Planck-Institut für Kognitions- und Neurowissenschaften, Leipzig, Germany
eichelbaum@informatik.uni-leipzig.de

In the last years a variety of open-source software packages focusing on visualization of human brain data have evolved. Many of them are designed to be used in a pure academic environment and are optimized for certain tasks or special data. The open source visualization system we introduce here is called *OpenWalnut*. It is designed and developed to be used by neuroscientists during their research, which enforces the framework to be designed to be very fast and responsive on the one side, but easily extendable on the other side. *OpenWalnut* is a very application-driven tool and the software is tuned to ease its use. Whereas we introduce *OpenWalnut* from a user's point of view, we will focus on its architecture and strengths for visualization researchers in an academic environment.

6.1 Introduction

The ongoing research into neurological diseases and the function and anatomy of the brain, employs a large variety of examination techniques. The different tech-

niques aim at findings for different research questions or different viewpoints of a single task. The following are only a few of the very common measurement modalities and parts of their application area: *computed tomography* (CT, for anatomical information using X-ray measurements), *magnetic-resonance imaging* (MRI, for anatomical information using magnetic resonance esp. for soft tissues), *diffusion weighted MRI* (dwMRI, for directed anatomical information for extraction of fiber approximations), *functional MRI* (fMRI, for activity of brain areas indicated by the blood-oxygen-level dependence (BOLD) effect) and *electroencephalography* (EEG, for activation of certain brain areas indicated by electric fields).

Considering the different applications, it is evident that, for many research areas, only a combination of these techniques can help answering the posed questions. To be able to analyze data measured by the different techniques, a tool that can efficiently visualize different modalities simultaneously is needed. The software (called *OpenWalnut*) we present in this paper aims at exactly this task. It does not only allow to display different modalities together but also provides tools to analyze their interdependence and relations.

Throughout the paper, we describe the general software architecture, its interactive multi-modal visualization capabilities, and how these make it especially suitable for the task of multi-modal analysis of measurements of the human brain. To obtain a first overview of the context we review some related software.

6.1.1 Related Software

There exist several visualization packages that are similar to *OpenWalnut* in some aspects or that are designed for similar application areas as *OpenWalnut*. Of the packages we are aware of, MeVisLab ([MVL, 2010]) and Amira ([Amira, 2010]) are the programs that come closest to *OpenWalnut*. Both are based on the principle of data flow networks and provide a *graph widget* that allows the user to manipulate this graph directly. While there is a free version of MeVisLab which provides a large subset of the tool's rich feature set, there exist three different variants of Amira. In addition to the two commercial variants of Amira that slightly differ in focus, there is an academic version developed at the Zuse Institute Berlin, which is freely available for collaborating institutes.

Another open-source tool for visualization of biomedical data is developed at the Scientific Computing Institute (SCI) at the University of Utah. It is part of a larger framework for simulation and visualization called SCIRun ([SCIRun, 2010]). Similar to Amira and MeVisLab it is based on a data flow network.

A major difference of *OpenWalnut* compared to these tools is the visibility and use of the data flow network (called *module graph* in *OpenWalnut*) to users. As the complexity of module graphs can grow very fast, its construction yields a fast increasing barrier for the user. In contrast to other open-source tools (like

MeVisLab and SCIRun), *OpenWalnut* can hide this complexity completely from the user and is, therefore, also suitable for scientists who simply want to use visualization tools for their data but are not familiar with or do not want to deal with the visualization internals. SCIRun provides so-called *power apps* to hide this complexity of the data flow environment. These, and similar macros in MeVisLab are very helpful to provide simple user interfaces for special tasks. However, they still have to be created with a script that uses the network in the background. Later in this paper we will describe how *OpenWalnut* combines the best out of two worlds: on the one hand, it provides an easy-to-use graphical user interface (inspired by ParaView [Ahrens et al., 2005]), making it a plug-and-play visualization tool. On the other hand, it provides an optional direct access to the data flow network.

Another package for analyzing imaging data of the human brain is FSL ([FSL, 2008]). It consist of a number of loosely coupled tools and a main GUI for starting sub-tools that serve different tasks like image registration, image visualization, and segmentation. The last application we want to mention here is MedINRIA ([MedINRIA, 2009]), which also provides modules for brain visualization, fiber tracking, and processing of tracking data. All of these tools are integrated in a common windows as user interface, where the window adapts to the chosen task.

Another approach has been chosen by Kindlmann in the *teem* ([teem, 2009]) library. It is not a visualization tool in the sense that it does not provide an interactive graphical user interface, but rather provides a large number of algorithms that are useful for the analysis and visualization of medical imaging data. Its command-line interface communicates through pipes and allows to build simple visualization pipelines and store intermediate data as well as final images in files. As it provides a C interface as well and as it is published under a free software license, other tools, similar to *OpenWalnut*, can benefit directly from teem's data processing capabilities.

The choice for developing *OpenWalnut* from scratch came out of the needs of the neuroscientists at the Max Planck Institute for Human Cognitive and Brain Sciences (MPI CBS) in Leipzig: For their research, they wanted an open-source tool, that is usable for people not familiar with data flow networks and allows for multi-modal visualization of the human brain in a single, coherent environment. Unfortunately, none of the above mentioned tools could fulfill all these needs for them.

Finally, it should be mentioned that there exist even more visualization tools that have a somehow similar approach concerning the user interface but are fitted to other user communities (i.e. not bio/neuro/medical). Examples are Vish [Benger et al., 2007], Mayavi [Enthought Inc., 2010], and the very popular ParaView [Ahrens et al., 2005].

6.2 Design and Architecture

OpenWalnut's design was mainly steered by two criteria: Firstly, it has to be a powerful and easily expandable framework for visualization researchers allowing them to implement algorithm prototypes and ideas quickly and easily while, secondly, providing an intuitive graphical user interface for neuroscientist researchers who include *OpenWalnut* in their daily research tasks. Whereas the first criterion asks for a flexible and extendible framework, the second criterion introduced the need for a high level of interactivity and responsiveness of the application.

To achieve these ambitious goals, it is important to split functionality and interface. Known and famous in the context of object-oriented programming ([Gamma et al., 1994]) this principle allows a powerful and complex framework under the hood of a simple interface, the GUI in our case. This way, the addition of new modules to the system does not require any changes in the GUI or other parts of the software as they are integrated using abstract interfaces and provide their parameters, settings, and I/O information using a standardized interface. To furthermore allow the user to modify data, tune algorithm parameters, or simply load and execute new algorithms while other algorithms are running, a multi-threaded approach is nearly unavoidable. Modules and parts of the basic framework should be able to work independent of each other. We avoid the *data pull* principle, also known as polling, wherever possible, because it causes many synchronization issues. We strongly focus on the push approach. Thus, data changes are not queried by parts of the program, but they are propagated automatically whenever a change occurs. Figure 6.1 illustrates the architecture of our medical visualization system called *OpenWalnut* and its algorithm-centric layout which is illustrated in the next section in more detail.

6.2.1 Architecture

This section covers the details of the software architecture, which is shown on an abstract level in Figure 6.1, and its implementation. The core parts provide the graphics engine, all basic data handling facilities, and basic mathematical and utility functions to the modules. The *kernel* with the *modules* and the module management is stacked on top of it and provides the actual interface implementation used to map the module graph structure and its properties to an end-user interface. From the module-programmer's point of view, the framework is designed entirely based on the aim to make programming easy and to enable us to achieve results in a minimum of time. The data structures provided for module parameters and data exchange allow the modules to provide information using data and parameters in a very abstract fashion without any knowledge of the actual user interface or other modules in the module graph. Therefore, the programmer can

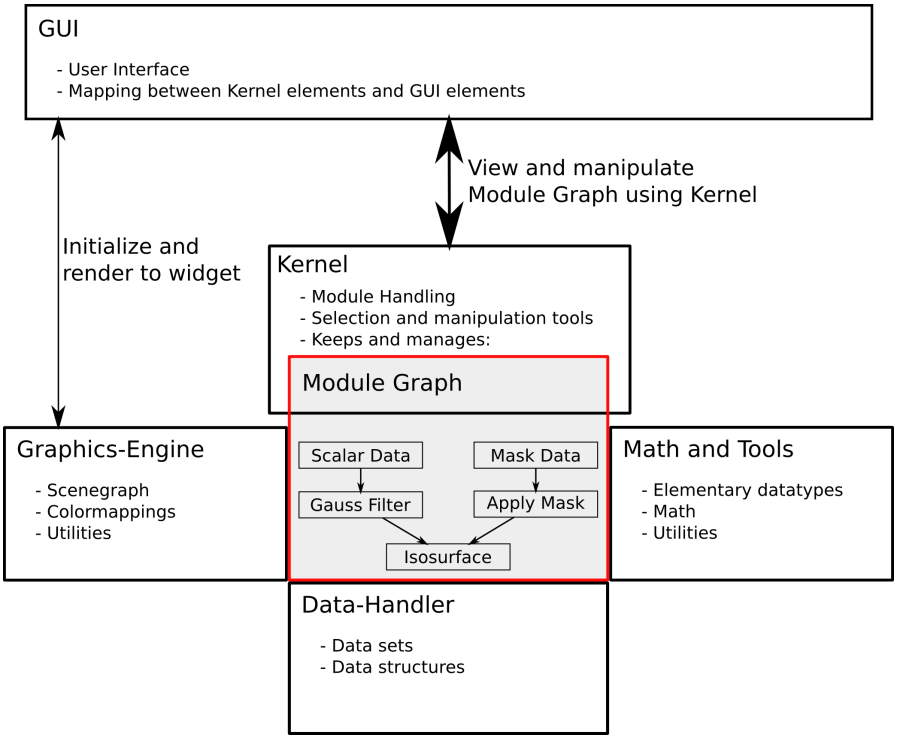


Figure 6.1: Software architecture. The graphical user interface sits on top of the kernel and maps the module graph and its properties to GUI elements. The modules utilize the core functionality and are handled by the kernel.

focus on module programming, only; no boilerplate code is needed for any kind of GUI interaction or graphics setup.

Graphics Engine

Starting bottom-up, let us first have a closer look at the graphics engine. The graphics engine mainly provides an interface to OpenSceneGraph ([OSG, 2010]). OpenSceneGraph is designed to be used in multi-threaded environments and provides most of the tools and structures required for creating and modifying graphics data. It contains tools to manage large triangle meshes, textures, shaders, and encapsulates most current features of OpenGL ([Khronos, 2010]). This ensures a maximum of flexibility during module development. One example of our extensions of OpenSceneGraph in the *Graphics Engine* are flexible color maps: They can simply be added to any other kind of rendering data, be it geometrical data or not. It automatically manages the loaded data volumes used for color-mapping, their ordering, blending factors, or the actual color map of each volume. This helps the module programmer to provide surface coloring in a clean and straight-forward way and fulfills the basic requirement to quantitatively analyze data.

Data Handler

The data handler provides the different kinds of data sets and data structures. Besides this, the data handler provides supportive algorithms for analyzing data or spatial partitioning of volume data. For example, the data handler provides the tools to convert and scale volume data to be used as texture. These textures and scaling information are then used by the color-mapping facility in the graphics engine to provide proper color mapping for loaded data on arbitrary graphical scene graph elements.

Due to the strong focus of *OpenWalnut* towards medical, especially neurological data, most volumetric data is stored using an implicitly defined grid. Besides this regular three-dimensional data, other kinds of grid structures exist. *OpenWalnut* provides an abstract kind of grid which can be used to implement nearly all other kinds of grid structures. As the grid is stored implicitly, additional transformations are stored inside the grid to ensure that the orientation of a volume in space is right according to the dataset or previously applied registration algorithms.

Additionally, the data handler provides the input and output interfaces to read and write datasets from and to files.

Kernel and GUI

The core component of *OpenWalnut* is the *kernel*. It accommodates running modules, organizes them in a data flow network (or *module graph*), and handles all

operations thereon. The most interesting part in the kernel is how modules get integrated into the system. Generally, modules do not have any knowledge about the GUI or other modules in the graph and run in their own thread. Besides this, the literal meaning of graphical user interface might confuse at this point. The task of the GUI, in our case, is mainly to provide the interface to the kernel and to the module graph including its properties. This might be a real graphical representation as *OpenWalnut* provides but can also be a simple command-line interface or a script interface that do not provide graphical output. This is essentially possible due to the abstract command-like interfaces provided in the kernel. In the next paragraph, we will introduce the module graph, the module's communication possibilities, and how this interacts with the GUI.

A module has exactly one possibility to interact with other modules residing in the kernel. Modules can define so called *connectors*. These connectors define the input and output channels of a module and define the exact type of data this connector supports. This ensures that modules always get the right kind of data to the correct input, are notified of changes to their input, and can update their output data which may be intermediate data structures or graphics stored in the scene graph. This way, changes in one module propagate along the graph and wake up directly depending modules allowing them again to process the new data. The typed connectors allow the kernel to decide whether module inputs match to a module output. The kernel uses this to provide facilities to the GUI to get lists of compatible connections in other modules or other connection possibilities. The kernel uses this information to create a so called *combiner* which represents the connection or module creation request. This combiner is the abstraction used by the GUI to display those options. As each module and connector provides its name, icon, and description text, the GUI can automatically create a button or menu entries using this information. As module instantiation can take a while, especially if a module does some costly initialization, applying a combiner is done asynchronously. While the kernel processes a combiner, it uses callbacks to inform the GUI about its progress and possible state changes. As mentioned earlier, the whole architecture is designed to use the push mechanism to propagate data, states, and other information. The kernel makes heavy use of this and provides callback and signaling mechanisms for nearly all possible operations. This ensures that the kernel does not need to be polled somehow.

The remaining task is the communication of module parameters and other settings to the user. During the design process, we always avoided that modules have to know the GUI or need to specify their GUI representation directly. Modules therefore are equipped with a mechanism called *properties*. These properties enabled the module developer to simply define parameters or settings without any knowledge of their representation. A module can, for example, define a property

of type `double` with a name and a description associated with it. In addition, it can define constraints, for example that it only accepts positive values, to exactly define valid values. The interesting part here is, that it is up to the GUI to decide about the graphical representation of those parameters. To stick with the double-precision floating-point property example, the GUI can decide whether a slider or a text box is more appropriate for a property depending where it is displayed. It therefore mainly uses the constraints defined on a property. Another example would be a property representing a four-times-four matrix that can be represented by sixteen text fields or by several sliders defining rotation, scaling, and translations. Whenever the user modifies a property in the GUI, the property automatically checks whether the new value is valid by using the before-mentioned property constraints. If the value is invalid, the property rejects it and the GUI can somehow show it to the user. If the value is valid, it gets set for the property and the automatic change-propagation ensures that all observers, especially the module owning the property, are notified about the value change. A module can then wake up from its sleep state to handle the new value. It is also possible to use these properties directly in scene graph nodes in conjunction with OpenSceneGraph's callback mechanism to directly modify graphical entities. As the properties implement the *observable pattern*, they can be used in a variety of ways.

With an increasing amount of fine-grained algorithm implementations in modules, the complexity of needed GUI interaction increases tremendously if results of algorithms need to be reused as input for other modules. To circumvent the problem, *OpenWalnut* provides *module containers*. These containers can accommodate multiple modules and forward their connectors and properties. This allows a module container to look and behave like a normal module. The module programmer can re-combine modules in a certain way to map a work-flow or visualization use-case without revealing the underlying complexity. This, on the one hand, hides complexity from the user and, therefore, makes the software more intuitive and, on the other hand, allows programmers to reuse existing modules and algorithms.

6.2.2 Control Panel — Hiding the Module Graph

As mentioned above, *OpenWalnut* hides its module graph by default. Instead, user interaction is performed using a tree widget in the control panel (right in Figure 6.2). While this tree still allows complete access to the module graph for advanced users (not described in this paper), it provides a simplified interface to *OpenWalnut's* functionality for users who are not interested in the graph. The tree widget acts similar to that of ParaView [Ahrens et al., 2005]: In the first level of the tree the loaded data sets are shown. The branches below indicate the modules that were applied to the data or other modules in higher levels of the tree. To adjust

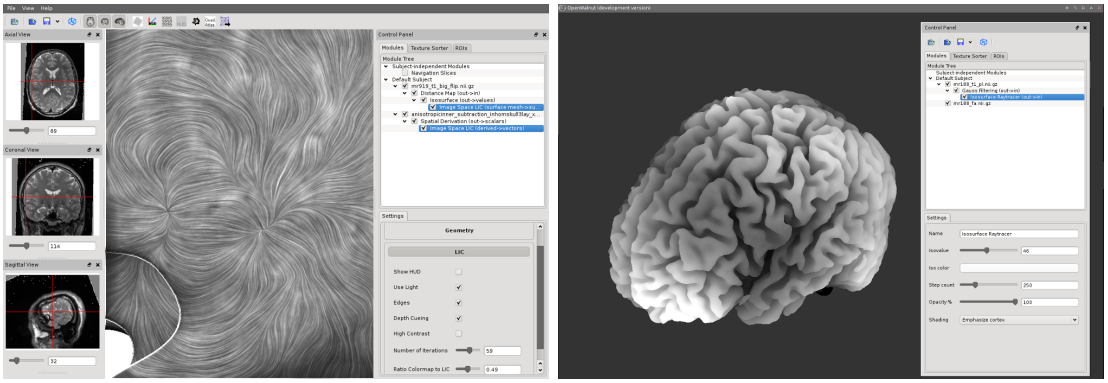


Figure 6.2: Left: *OpenWalnut's* default GUI. It mainly is split in three areas. The navigation windows on the left showing axial, coronal and sagittal slices through the anatomy which allow easy orientation inside the data. The main 3D view contains the scene itself. Most of the user-interaction with several modules and data can be done in the control panel on the right. It represents how the modules are wired to each other with their connectors and provides an intuitive panel for changing a module's properties. Right: The GUI can be highly customized.

settings of a specific module, a user selects the module and the properties of the module appear in the "Settings" widget below. Clicking onto a module also has a second use: It updates the toolbar (top row in Figure 6.2) to show only the modules whose input connectors fit the output connectors of the selected module. Thus, the toolbar always presents all currently applicable modules to the user. Clicking an icon in the toolbar adds the corresponding module in a branch below the currently selected module in the tree and executes the module. To ensure consistency, only leafs in the tree can be removed. This ensures that no module looses the modules it depends on. Removing a module undoes all effects that have been caused by the module.

6.3 Results and Conclusion

Today, *OpenWalnut* is a visualization tool heavily used by the Max Planck Institute for Human Cognitive and Brain Sciences and the Max Planck Institute for neurological research. Besides the usage as a pure visualization tool, it is also a powerful and handy framework for visualization researchers. In this paper, we gave an overview of *OpenWalnut's* architecture and design which mostly is interesting from the visualization researcher's point of view. We have shown that *OpenWalnut* uses several abstract methods to allow modules to communicate and process data. For visualization researchers, this minimizes efforts and allows them to focus on

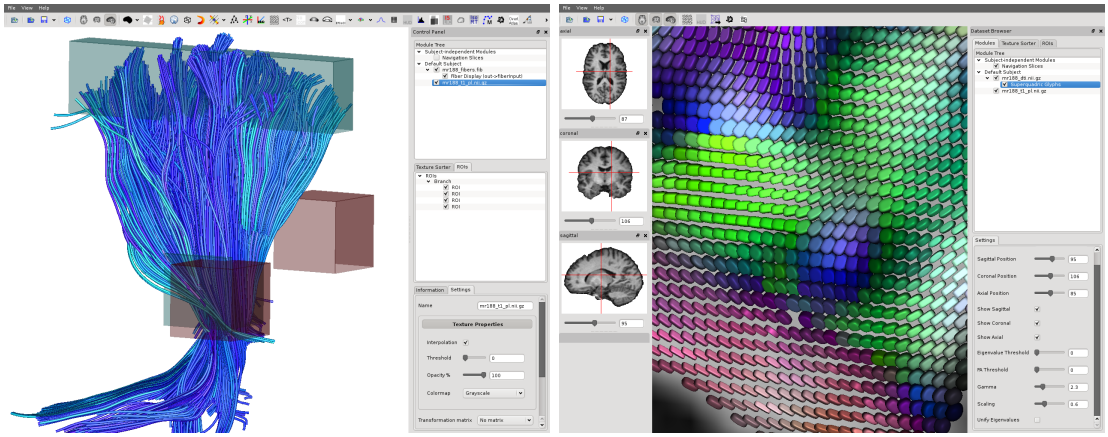


Figure 6.3: Two screenshots showing *OpenWalnut's* selection tools. On the left, regions of interest (ROI) have been used to select a part of the corticospinal tract of a fiber-tract dataset. On the right, the navigation slices have been used to select and view a slice in a DTI dataset in which superquadric tensor glyphs are shown ([Hlawitschka et al., 2008]).

development of novel algorithms.

The module-graph allows the arbitrary combination of modalities even among multiple subjects, which makes it a very handy tool for neuroscientists during their research tasks for inter-subject and/or multi-modal analyses. The graphical user interface only provides the editing features for the module graph and can provide very distinct GUI elements for the module graph representation or the representation of module parameters and settings. This way, the GUI can be seen as a generalized *view* on top of the functionality of *OpenWalnut*. The GUI itself is designed to be clean, structured and easy to understand but it provides advanced elements for experienced user.

As *OpenWalnut* is completely open source, it can be used, extended, and customized by everyone to fulfill their needs and provides a framework for testing and implementing algorithms in a very easy way. Unlike other software tools, the strict coding standard and documentation standard ensures a consistent code style and a very detailed documentation of the involved classes. An extensively documented example module helps developers to directly start programming own modules. It is ideal for researchers in the area of visualization and a nice and intuitive tool for visualization users.

The website `http://www.openwalnut.org` provides a lot of information, screenshots, and source code.

Acknowledgments

The authors are grateful for the advice, help and ideas of Christian Heine, Mathias Goldau, Alfred Anwander, and Thomas Knösche. This work was supported by AIF (ZIM grant KF 2034701SS8), NSF CCF-0702817, and NIH R21EB009434-01S1.

Bibliography

[Ahrens et al., 2005] Ahrens, J., Geveci, B., & Law, C. (2005). Paraview: An end-user tool for large data visualization. In C. Hansen & C. Johnson (Eds.), *Visualization Handbook*. Elsevier.

[Amira, 2010] Amira (2010). Amira - visualize analyze present. http://www.amira.com/.

[Benger et al., 2007] Benger, W., Ritter, G., & Heinzl, R. (2007). The concepts of vish. In 4^{th} *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007* (pp. 26–39).: Berlin, Lehmanns Media-LOB.de.

[Enthought Inc., 2010] Enthought Inc. (2010). Mayavi - 3d scientific data visualization and plotting. http://code.enthought.com/projects/mayavi/.

[FSL, 2008] FSL (2008). FMRIB software library. http://www.fmrib.ox.ac.uk/fsl/.

[Gamma et al., 1994] Gamma, E., Helm, R., & Johnson, R. E. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman.

[Hlawitschka et al., 2008] Hlawitschka, M., Eichelbaum, S., & Scheuermann, G. (2008). Fast and memory efficient GPU-based rendering of tensor data. In *Proceedings of the IADIS International Conference on Computer Graphics and Visualization 2008* (pp. 36–42).

[Khronos, 2010] Khronos (2010). The OpenGL Graphics System: A Specification. URL: `http://www.opengl.org`.

[MedINRIA, 2009] MedINRIA (2009). http://www-sop.inria.fr/asclepios/software/MedINRIA/.

[MVL, 2010] MVL (2010). MeVisLab - development environment for medical image processing and visualization. http://www.mevislab.de/.

[OSG, 2010] OSG (2010). OpenSceneGraph. URL: `http://www.openscenegraph.org/`.

[SCIRun, 2010] SCIRun (2010). SCIRun: A scientific computing problem solving environment, scientific computing and imaging institute (SCI). http://www.scirun.org.

[teem, 2009] teem (2009). Teem: Tools to process and visualize scientific data and images. http://teem.sourceforge.net/.

Article 7

Implementation of an Algorithm for Approximating the Curvature Tensor on a Triangular Surface Mesh in the Vish Environment

Edwin Mathews[1], Werner Benger[1], Marcel Ritter[2]
[1]Center for Computation & Technology
at Louisiana State University (CCT/LSU), Baton Rouge, Louisiana, USA
[2] Unit of Hydraulic Engineering,
Department of Infrastructure, University of Innsbruck

Finding local surface properties of a generated mesh is an essential component in applications across several fields. Specifically, Gaussian curvature provides intrinsic geometric information of local shape characteristics on a surface. It finds use in mesh applications like 3-D scanned image noise smoothing, feature recognition, and data analysis. An algorithm was developed for the Vish environment to approximate the shape operator from the curvature tensor using only lists of triangle vertex positions and individual vertex positions. The algorithm is based on a curvature tensor approximation method developed by Gabriel Taubin and does not need information about the mesh edges to be provided explicitly in the calculation. From the curvature tensor, the principle directions and curvatures can be found and used to calculate the Gaussian curvature and mean curvature at each vertex. Using this information, an application is given where the curvature is used to analyze mixing on the surface of a fluid 'virtual bubble.'

7.1 Introduction

Development of a Vish-specific algorithm that takes a mesh surface as an input and returns a field of curvature values is an important foundation in the development of new visualization modules. In current practice, curvature calculation is essential to feature detection and noise filtering of sampled three dimensional geometry and to the detection and analysis of medical images. With the broad scope of polyhedral surfaces in computer graphics in industrial and biomedical engineering, robotics, and several other fields, it is a start for where Vish may find itself in years to come. Here, curvature finds a novel use in the analysis of Computational Fluid Dynamics (CFD) time surfaces [Bohara et al., 2010]. The time surface is a virtual bubble constructed from seed points in a CFD generated vector field [Benger et al., 2009]. It can be seen as a higher dimensional extension of timelines, where a continuous mesh surface constructed from the seed points evolves over time. Being able to visualize a changing surface in a large data model has advantages in observing fluid movement and mixing over timelines. Where timelines are effective in displaying the trajectory and rotation of a particle, time surfaces give a more intuitive visualization of the effects of the time dependent vector field and of the movement of multiple particles in relation to each other.

7.1.1 Application in Analysis

The time surface was intially developed to visualize a simulation of stirred tank data to help improve mixing efficiency of the stirring process [Bohara et al., 2010]. Without a means to quantify the mixing, conclusions about the effectiveness of the stirred tank had to be made via visual inspection of the time surface. In general, mixing is the combining of two or more substances into one mass with a thorough blending of the constituents. So how can we quantify this?

Surface shape information at each vertex is useful in representing the mixing on the time surface because the variation of a vertex position relative to its neighboring vertices can be quantified. If the curvature of a vertex is initially zero then changes in magnitude abruptly, it signifies a quick movement of the particles around the vertex in relation to its neighbors. Unfamiliar particles will then come into contact as a result of replacing the particles that have just moved away. So while the curvature itself does not represent the mixing of the fluid locally, but the change of curvature does. When evaluating over a finite mesh, it is important to compute intrinsic information about the surface so that nominal values of the information will not be dependent on the mesh refinement. The Gaussian curvature is an intrinsic invariant of the surface as opposed to the extrinsic invariant mean curvature; it does not depend on its embedding, and a mesh approximation will only become more accurate with a higher refinement.

7.2 Mathematic Principles

In this application, the method looks to find the shape operator at some point p defined on a surface S. The shape operator, or Weingarten map, is a type of extrinsic curvature that is a linear operator on the tangent space at each point p on the surface S. At any point, it is equal to the Jacobian of $\mathbf{N}$, the function that yields the unit vectors normal to the surface. To calculate this on a grid vertex, the contribution from each edge attached to the vertex must be averaged. Along this edge where the normal vectors at two adjacent points are given, the difference of the normal vectors $dN^k = N_0^k - N_1^k$ with respect to each coordinate direction $dx^i = x_0^i - x_1^i$ can be found. In three dimensions,

$$J_i^k = \begin{pmatrix} \frac{dN}{dx^1} \\ \frac{dN}{dx^2} \\ \frac{dN}{dx^3} \end{pmatrix} = \begin{pmatrix} \frac{dN^1}{dx^1} & \frac{dN^2}{dx^1} & \frac{dN^3}{dx^1} \\ \frac{dN^1}{dx^2} & \frac{dN^2}{dx^2} & \frac{dN^3}{dx^2} \\ \frac{dN^1}{dx^3} & \frac{dN^2}{dx^3} & \frac{dN^3}{dx^3} \end{pmatrix} = \frac{dN^k}{dx^i} \tag{7.1}$$

where J_i^k is the Jacobian along this edge vector given by $x_0 - x_1$. The projection of the surface Jacobian on to the tangent plane can be computed if two vectors tangent to the surface, $\vec{u}$ and $\vec{v}$, are given where $\vec{u}$, $\vec{v}$, and the normal vector at p, form an orthonormal basis on the surface. This shape operator in the surface is equal to

$$S = \begin{pmatrix} J(u,u) & J(u,v) \\ J(v,u) & J(v,v) \end{pmatrix} = \begin{pmatrix} \kappa_p^{11} & \kappa_p^{12} \\ \kappa_p^{21} & \kappa_p^{22} \end{pmatrix} \tag{7.2}$$

where the operation $J(\,\cdot\,,\cdot\,)$ is of the form $J(u,u) = u^t J u$ and $J(u,v) = u^t J v$. The shape operator can also be defined using the Metric Tensor, $I = I_{xx}dx^2 + 2I_{xy}dxdy + I_{yy}dy^2$, and the Extrinsic Curvature, $II = II_{xx}dx^2 + 2II_{xy}dxdy + II_{yy}dy^2$, coordinate description of the tangent space at the surface S at p. Using the coefficients of the first and second fundamental forms

$$S = \frac{1}{det\ I} \begin{pmatrix} II_{xx}I_{yy} - II_{xy}Ixy & II_{xy}I_{yy} - II_{yy}I_{xy} \\ II_{xy}I_{xx} - II_{xx}Ixy & II_{yy}I_{xx} - II_{xy}I_{xy} \end{pmatrix}$$

where $det\ I = I_{xx}I_{yy} - I_{xy}^2$, the area of the surface element. The result for both formulations is a 2×2 symmetric matrix where the eigenvalues are just the principle curvatures at some p on S. Accordingly, the determinant is the Gaussian curvature and half the trace of the shape operator is the mean curvature. To find the directional curvature at some point with a defined shape operator in direction $\vec{T}$ defined in surface coordinates, the following operation is performed

$$\kappa_p = S(T,T) = \begin{pmatrix} t_1 & t_2 \end{pmatrix} \begin{pmatrix} \kappa_p^{11} & \kappa_p^{12} \\ \kappa_p^{21} & \kappa_p^{22} \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \tag{7.3}$$

where $\vec{T} = t_1\hat{T}_1 + t_2\hat{T}_2$ is a tangent vector on S at p. $\hat{T}_1$ and $\hat{T}_2$ are the principle directions on S at p when the shape operator is diagonal and forms an orthonormal basis of the tangent space. If the normal vector N at p is added to this orthonormal basis, eqn. (7.3) is extended to non-tangential directions

$$\kappa_p = \begin{pmatrix} n & t_1 & t_2 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & \kappa_p^{11} & \kappa_p^{12} \\ 0 & \kappa_p^{21} & \kappa_p^{22} \end{pmatrix} \begin{pmatrix} n \\ t_1 \\ t_2 \end{pmatrix} \tag{7.4}$$

and $\vec{T}$ is expanded to $\vec{T} = n\hat{N} + t_1\hat{T}_1 + t_2\hat{T}_2$. The vector $\vec{T}$ can be written as a linear combination of another orthonormal system. The directional curvature can still be evaluated using this different coordinate system with another 3×3 symmetric matrix, U_p, as long the three eigenvalues are still 0, κ_p^{11}, and κ_p^{22}. The shape operator can be recovered from the curvature tensor U_p by restricting the matrix to the tangent plane.

Using the method described by Taubin [Taubin, 1995], the curvature tensor is approximated by defining a matrix M by an integral formula that has the same eigenvectors as U_p and has their eigenvalues related by a fixed homogeneous linear transformation. For $-\pi \leq \theta \leq \pi$ on the tangential plane, $\vec{T_\theta}$ at p is the unit length tangent vector $\vec{T_\theta} = cos(\theta)\hat{T}_1 + sin(\theta)\hat{T}_2$. $\hat{T}_1$ and $\hat{T}_2$ represent the same orthonormal principle directions in surface coordinates as previously mentioned. Using this in the expression for directional curvature:

$$\kappa_p = \begin{pmatrix} \cos\theta & \sin\theta \end{pmatrix} \begin{pmatrix} \kappa_p^{11} & \kappa_p^{12} \\ \kappa_p^{21} & \kappa_p^{22} \end{pmatrix} \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$$

Because $\hat{T}_1$ and $\hat{T}_2$ are the principle directions, $\kappa_p^{12} = \kappa_p^{21} = 0$. Evaluating the equation results in $\kappa_p(T_\theta) = \kappa_p^{11}cos^2(\theta) + \kappa_p^{22}sin^2(\theta)$. Integrating over the entire surface element about p, the 3×3 approximation matrix M is defined as

$$M = \frac{1}{2\pi}\int_{-\pi}^{\pi} \kappa_p(T_\theta)(T_\theta \otimes T_\theta)d\theta \tag{7.5}$$

In M, the normal vector N is an eigenvector associated with the zero eigenvalue since $T_\theta \otimes T_\theta$ is a rank 1 matrix at every θ and $\vec{T_\theta}$ is tangent to the surface at p. If M is then factorized and and integrated, it can be shown that the eigenvalues of M, m^{11} and m^{22}, are related to the principle curvatures by

$$\kappa_p^{11} = 3m_p^{11} - m_p^{22} \tag{7.6}$$

$$\kappa_p^{22} = 3m_p^{22} - m_p^{22} \tag{7.7}$$

To approximate the curvature κ_p along some tangential vector $\vec{T}$, we use a curve $q(s)$, a normal section to S at p parameterized by arc length. By differentiating and solving at $s = 0$, $q(0) = p$, $q'(0) = T$, and $q''(0) = \kappa_p(T)N$, where N is the unit length normal vector. When $q(s)$ is expanded in Taylor series

$$q(s) = q(0) + q'(0)s + \frac{1}{2}q''(0)s^2 + O(s^3)$$

$$= p + Ts + \frac{1}{2}\kappa_p(T)Ns^2 + O(s^3)$$

Removing higher terms and solving for $\kappa_p(T)$,

$$\kappa_p(T) = \lim_{s\to 0} \frac{2N^t(q(s) - p)}{||q(s) - p||^2} \approx \frac{2N^t(p_j - p_i)}{||p_j - p_i||^2} \tag{7.8}$$

where p_j is another point on the surface close to the point p_i, where the approximation is being done. This allows for the evaluation of the directional curvature directly on a mesh surface between two neighboring points p_j and p_i. Now, all of the components needed to compute the shape operator have been defined. The integral formula for M when solving at p_i is discretized to

$$\tilde{M}_i = \sum_{j\in i} w_{ij}\kappa_{ij}(T_{ij} \otimes T_{ij}) \tag{7.9}$$

where j are the vertexes surrounding vertex i. The weighted value w_{ij} is value based on the area of the triangles bordering the tangent vector. Once $\tilde{M}_i$ is calculated, a Householder matrix Q corresponding to the plane orthogonal to the unit normal vector is used to decompose the 3×3 matrix $\tilde{M}$. The Householder reflection is a transformation that describes a reflection by a plane, in this case the tangential plane at p on S. By construction, the first column of the Householder matrix is equal to the unit normal vector and the other two columns are an orthonormal basis of the tangent space. Because the unit normal vector at p_i is an eigenvector of $\tilde{M}_i$ corresponding to the 0 eigenvalue,

$$Q^t\tilde{M}_iQ = \begin{pmatrix} 0 & 0 & 0 \\ 0 & m_p^{11} & m_p^{12} \\ 0 & m_p^{21} & m_p^{22} \end{pmatrix} \tag{7.10}$$

The values of the 2×2 minor can then be used to find the values of the principle curvature using the relations above for κ_p^{11} and κ_p^{22}. From there, as described above, the Gaussian curvature or mean curvature can be evaluated at that vertex.

7.3 Our Approach

To use the algorithm, an input list of vertices and list of triangles with the three vertices that make up each triangle of the current grid are used. Instead of requiring a list of edges on the grid surface, they will be computed for each triangle. The first step is to compute the weighted unit normal vector, N_{v_i}, of each vertex.

$$N_{v_i} = \frac{\sum_{f_k \in F^i} N_{f_k} A_k}{||\sum_{f_k \in F^i} N_{f_k} A_k||} \tag{7.11}$$

To find N_{v_i}, the normal vector, N_{f_k}, of each triangle face f_k surrounding vertex i is multiplied by its face area A_k. The sum of all faces is then normalized. This is done within a loop for all triangles on the surface. First, coordinates for each of the three vertices in the triangle are needed.

```
for (i = 0; i <Number of Triangles; i++)
  {
      point A, B, C;
      A = ... ; B = ... ; C = ... ;
```

Then, two of the triangle edge vectors are computed for finding the area and normal vector.

```
      tvector BA = B - A;
      tvector CA = C - A;
```

The area is determined by half of the cross product norm of the same two edge vectors. The triangular area is placed into an array.

```
      TriangleArea[ i ] = .5*norm(BA ^ CA);
```

The weighted triangle surface normals are then weighted by their area and placed in a 3-component vector array.

```
      WeightedTriangleNormal[ i ] = (BA^CA);
```

A nested loop is created for each of the vertices in the triangle. The area of the triangle is then summed to an array on the vertices. This `VertexArea` is a scalar equal to the area of all triangular faces surrounding a single vertex. It finds use later in the algorithm when calculating a weighted value and could also be visualized on the time surface to show where the surface is 'stretching'.

```
for(k = 0; k<3; k++)
  { VertexArea[T[k]] += TriangleArea[ i ];
```

Here, `T[k]` is the index of the three vertices in the triangle. For the call `Vertices Area[T[k]] += ...`, the area is being summed to the vertices through the current triangle in the outer loop. The weighted triangle normal vector is then summed to another three-component vector array on each of the vertices in the same fashion.

```
      WeightedVertexNormal[T[k]] += WeightedTriangleNormal[i];
  }
}
```

Both loops are then closed. After this section has completed, each vertex will have an array with the sum of all the triangular areas surrounding it and an array with the sum of all the triangular surface unit normal vectors. A new loop for all vertices on the surface is created to normalize the summed weighted vertex normals.

```
for(v=0; v < Number of Vertices; v++)
  {  WeightedVertexNormal[v] = WeightedVertexNormal[v].unit(); }
```

Next, another loop for all triangles is used to compute the curvature tensor of each vertex. To start, edge vectors must be computed for the first edge.

```
for(i = 0; i < Number of Triangles; i++)
  {    tvector AB, BA;
    AB = ... ; BA = ... ;
```

For the computation of M at the two vertices attached at the ends of this edge, a tensor weight w_{ij} is used. This weight helps to balance the contribution of edges representing a larger portion of the area surrounding the vertex. For all edge contributions to a vertex, $\sum w_{ij} = 1$. The weight for this triangle's contribution to the shape tensor is equal to the area of the triangle divided by twice the area of all the triangles surrounding the vertex to which it is contributing.

```
     double Wij0 = TriangleArea[ i ]/(2*VerticesArea[T[0]]);
```

Here, w_{ij} is computed for triangle vertex A, or 0, denoted by the term `Wij0`. The projection of the edge vector $\vec{AB}$ on the tangential plane is calculated using

$$T_{ij} = \frac{(I - N_{v_i} \otimes N_{v_i})(\vec{AB})}{||(I - N_{v_i} \otimes N_{v_i})(\vec{AB})||} \tag{7.12}$$

where I is the 3×3 identity matrix and N_{v_i} is unit normal vector at the vertex i. This operation can be performed by the single function

```
tvector TijAB = Tijoperator( WeightedVertexNormal[T[0]], AB );
```

Then, for the same edge the directional curvature needs to be found using eqn. (7.8). A function is `Kijoperator()` used for that purpose.

```
double KijAB = Kijoperator( WeightedVertexNormal[T[0]], AB );
```

Now, all of the components are available to compute the contribution of edge $\vec{AB}$ to the approximation of the curvature tensor at vertex A. The tensor product of T_{ij} is computed, multiplied by the two scalar values κ_{ij} and w_{ij} and, finally, added to a 3×3 matrix array indexed for the particular vertex.

```
Matrix33 TijABTensorProduct = ... ;
CurvTensor[T[0]] += Wij0 * KijAB * TijABTensorProduct;
```

where `CurvTensor` is a 3×3 matrix array. This operational procedure is repeated for the other edge vectors in the triangle. Each vertex will have two components of the curvature tensor added to it from each triangle of which it belongs, so for each triangle in the loop above, six curvature tensor components will be computed. To continue with computing the shape operator from the curvature tensor, the loop for all triangles is ended and a new loop for all the vertices on the surface is started. To find the Householder matrix which is needed to restrict the tensor to the tangential plane, the value of the sign in the calculation of W_{vi} must be found:

$$W_{vi} = \frac{E_1 \pm N_{v_i}}{||E_1 \pm N_{v_i}||} \tag{7.13}$$

where E_1 is the first coordinate vector and is equal to $(1, 0, 0)^t$. If the magnitude of $||E_1 - N_{v_i}|| > ||E_1 + N_{v_i}||$, the sign is negative; otherwise, it is positive.

```
}
for(v = 0; v < Number of Vertices; v++)
{ tvector E(1,0,0);
     tvector A = E + WeightedVertexNormal[ v ];
     tvector B = E - WeightedVertexNormal[ v ];
     double MagB = norm(B);
     double MagA = norm(A);
     tvector Wvi;
     if(MagB > MagA)
     { Wvi = B.unit(); }
     else
     { Wvi = A.unit(); }
```

The Householder matrix Q_{v_i} is then found by the formula $Q_{v_i} = I - 2W_{v_i}W_{v_i}^t$ And completing the restriction to the tangential plane, $Q_{v_i}^t \tilde{M}_{v_i} Q_{v_i}$ is computed.

```
Matrix33 QviT = ...;
Matrix33 Holder = QviT*CurvTensor[v]*Qvi;
```

where Holder is the final 3×3 Matrix whose first row and column are zeros. That leaves a 2×2 non-zero minor whose eigenvalues are m^{11} and m^{22} from eqn. (7.6) and (7.7). The two principle curvatures now finally allow the computation of the Gaussian or mean curvature of a triangular surface.

```
GaussCurve[ v ] = (3*m11 - m22) * (3*m22 - m11); }
```

7.4 Results

Using the curvature tensor module and pre-existing modules in Vish to render scalar and tensor data on a surface, we have been able to produce visualizations useful in the analysis of fluid mixing on the surface of time surfaces, such as illustrated in fig. (7.1). Here the Gaussian Curvature is shown on the triangular surface of a fluid blob of the stirred tank data set. The blue area identifies locations with a large positive Gaussian Curvature and red identifies a large negative value. As time elapses in the simulation, mixing is indicated by color change on the surface. As previously stated, this signifies the movement of particles within the surface and contact with new particles. Currently there are singularity points in the approximation related to a incorrect sign in the curvature. This tends to occur at points where the refinement of the mesh is very poor and less than six triangles are surrounding a vertex.

7.5 Future Work

The mesh refinement procedure for the time surface is presently being updated to improve the mesh quality, and may prove to diminish errors due to poor surface quality. Beyond that, values of the approximated curvature are being compared to surfaces with known curvature for validation and to estimate error. It is also of interest to directly compute the time derivative of the Gaussian curvature as a means of analysis instead of a visual interpretation. In future projects, the curvature module is looking to be used in the analysis of chemical dispersant mixing in Gulf of Mexico oil spill modeling. These disperants contain surfactants that dissipate oil slicks but are dependent on wave action and water movement for mixing. This modeling would utilize the time surface for visualization and could interpreted similarly to the stir tank data.

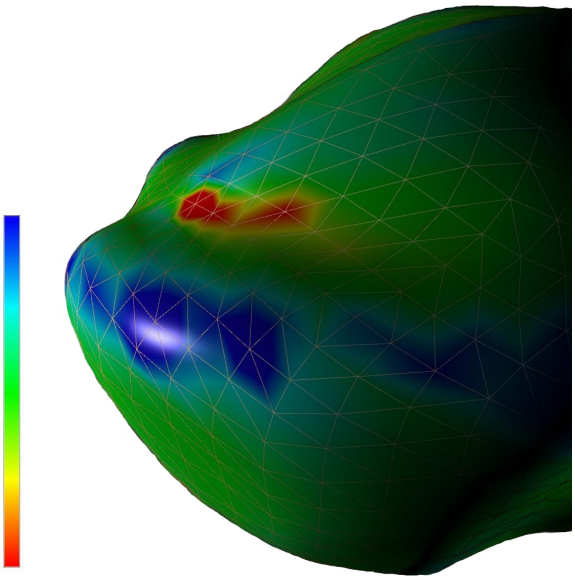


Figure 7.1: Gaussian Curvature on a triangulated time surface representing a fluid blob.

7.6 Summary

After presenting the mathematical foundation of surface curvature, an algorithm for its numerical computation was presented. The curvature tensor approximation was then implemented as a module in the Vish framework. In Vish, the module is currently being employed to compute the Gaussian curvature for the analysis of mixing on a CFD time surface. It provides a visually intuitive and quantifiable method of mixing evaluation in time surface simulations along with providing a characteristic value used in several other graphical applications.

Acknowledgements

Many thanks to the Group of Scientific Visualization at LSU and the Visualization Services Center at LSU for their assistance in the development of the algorithm. Thanks to the Louisiana Optical Network Infrastructure for providing the ability to contribute research in the field of scientific visualization and the Vish framework.

Bibliography

[Benger et al., 2009] Benger, W., Ritter, M., Acharya, S., Roy, S., & Jijao, F. (2009). Fiberbundle-based visualization of a stir tank fluid. In 17^{th} *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (pp. 117–124).

[Bohara et al., 2010] Bohara, B., Harhad, F., Benger, W., Brener, N., Iyengar, S., Ritter, M., Liu, K., Ullmer, B., Shetty, N., Natesan, V., Cruz-Neira, C., Acharya, S., & Roy, S. (2010). Evolving time surfaces in a virtual stirred tank. *Journal of WSCG*, 18(1-3), 121–128.

[Taubin, 1995] Taubin, G. (1995). Estimating the tensor of curvature of a surface from a polyhedral approximation.

Article 8

A Framework for Computing Integral Geometries in VISH using Template Meta Programming

Marcel Ritter[1,3], Bidur Bohara[1,2], Werner Benger[1]
[1]Center for Computation & Technology at Louisiana State University (CCT/LSU), USA
[2]Department of Computer Science at Louisiana State University (LSU), USA
[3]Unit for Hydraulic Engineering, Department for Infrastructure, Faculty of Civil Engineering at the University of Innsbruck, Austria
marcel.ritter@student.uibk.ac.at

We present a framework using C++ template programming for the computation of integral geometries such as streamlines, pathlines, geodesics or time surfaces. Hereby, common needs are identified and certain features are shared between different integration geometries minimizing the overall implementation effort. The implementation is based on a fiber bundle data model opening the possibility to handle all kinds of space-time geometries in an unified interface.

8.1 Introduction

Visualization of vector fields is still an topic of active research in scientific visualization. The most widely used approach, visualization via arrow icons, is intuitive but does not scale to huge datasets, where methods such as fastLIC [Stalling, 1998] or Doppler speckles [Benger et al., 2009b] are superior. Aside direct visualization

of the vectors at each data point a common approach is studying features of the vector field via integral curves and surfaces. The mathematical similarity of these feature indicators is not necessarily reflected by the actual implementation of the computation algorithms: the different integration scheme such as space-like vs. time-like or 1D vs. 2D may easily lead to independent implementations, whereas it were desirable to use one approach for all kinds of integral computations.

8.1.1 Mathematical Background and Motivation

Let $q(s)$ be a parameterized curve in a manifold M, $q : \mathbb{R} \to M : s \mapsto q(s)$. The variation of the curve parameter s defines the tangential vector $\dot{q} = dq/ds$ along the curve. In computational fluid dynamics (CFD) a vector field typically describes the motion of fluid particles. Let v be a vector field with the vector $v \in T_P(M)$ being an element of the tangential space at a point P of a manifold M. An integral curve is defined in a space-time manifold M by

$$\dot{q}(s) = v\big(q(s)\big) \quad \text{with} \quad q(0) = \sigma \in M \tag{8.1}$$

and σ the initial seeding point. Let us assume an evolving vector field in 3D coordinates. A **streamline** is an integral curve that is tangential to a vector field frozen at an instant of time, Fig. 8.1a:

$$\dot{q}(s) = v\big(q(0)^t, q(s)^x, q(s)^y, q(s)^z\big) \quad \text{with} \quad q(s)^t = q(0)^t \tag{8.2}$$

A **pathline** is evolving over time. It represents the motion of a fluid particle at a certain point in time, Fig. 8.1b:

$$\dot{q}(s) = v\big(q(s)^t, q(s)^x, q(s)^y, q(s)^z\big). \tag{8.3}$$

A bundle of integral curves yields a high dimensional object. An initial space-like seeding line, $\sigma : \mathbb{R} \to M : \lambda \mapsto \sigma(\lambda)$, defines an integral surface $\Sigma : \mathbb{R}^2 \to M : (s, \lambda) \mapsto \Sigma(s, \lambda)$:

$$d\Sigma/ds = v\big(\Sigma(s, \lambda)\big) \quad \text{with} \quad \Sigma(0, \lambda) = \sigma(\lambda) \tag{8.4}$$

A line at $\Sigma^t(s, \lambda) = const.$ is called a **material-line**, see Fig. 8.1c for an illustration. An initial seeding surface $S_0(\lambda, \mu) : \mathbb{R}^2 \to M : (\lambda, \mu) \mapsto S(\lambda, \mu)$ defines an integral-hyper-surface $H : \mathbb{R}^3 \to M : (s, \lambda, \mu) \mapsto H(s, \lambda, \mu)$:

$$dH/ds = v\big(H(s, \lambda, \mu)\big) \quad \text{with} \quad H(0, \lambda, \mu) = H_0(\lambda, \mu) \tag{8.5}$$

A surface at $H^t(s, \lambda, \mu) = const.$ is called a **time surface**, illustrated in Fig. 8.1d.

Finite differentiation schemes [Deuflhard & Bornemann, 2002] can be applied to solve these equations. In our visualization environment we take this common

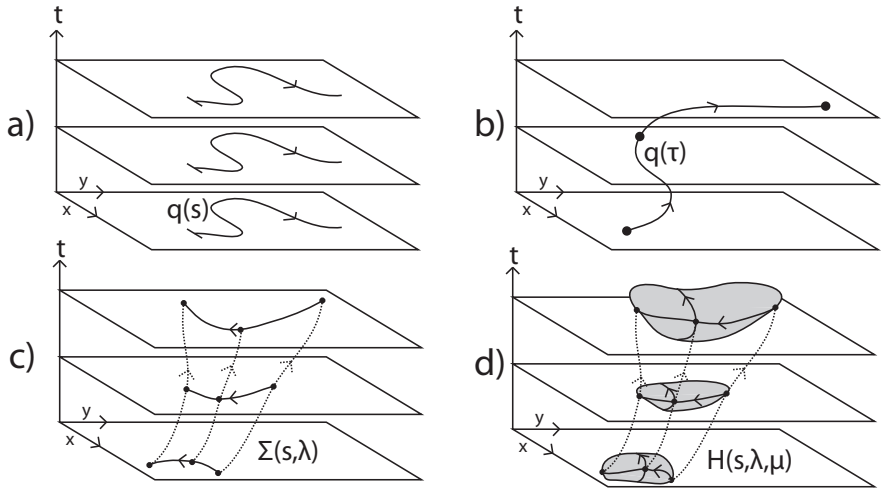


Figure 8.1: Different types of integral lines: a) streamlines, b) pathlines, c) material-line, d) time surface. The illustration shows time evolution in t-axis direction. The xy-plane is manifold hosting the integral curves.

foundation into account and we develop a template based integration framework suitable to all kinds of integration geometries in a unified way, sharing the algorithms used for solving differential equations, interpolating data fields and handling the underlying manifolds. The framework also extends to integrating geodesics on tensor fields stemming from numerical relativity or Magnetic Resonance Imaging (MRI). Here, the differential equation eq. 8.4 is replaced by the second order geodesic equation [Ritter & Benger, 2010] using $\dot{\Sigma} = d/ds\Sigma(s,\lambda)$:

$$\nabla_{\dot{\Sigma}}\dot{\Sigma}(s,\lambda) = 0 \quad \forall\lambda \quad \text{with} \quad \Sigma(0,\lambda) = \sigma(\lambda) \quad \text{and} \quad \dot{\Sigma}(0,\lambda) = v\big(\sigma(\lambda)\big) \tag{8.6}$$

8.1.2 Previous Work

New development in vector field visualization is related to the computation of streaklines and streaksurfaces [Weinkauf & Theisel, 2010], the invention of new integration geometry types, such as time surfaces [Krishnan et al., 2009] or by [McLoughlin et al., 2009] on the improvement of algorithms to compute stream and path surfaces .

8.2 Framework Design and Implementation

Our earlier work includes the development of a streamline module for the computation and rendering of streamlines. Later, a pathlines module was developed mostly independently, only sharing some vector field interpolation methods. For the streamline module, the computation part was separated from the rendering of lines and it was further generalized to ease the implementation of geodesics in higher dimensional space-times [Ritter, 2010], and the computation part of the pathline module was inherited to compute the time surfaces [Bohara et al., 2010]. Besides a fast and simple Euler integration we had implemented the DOP853 integration for high accuracy [Hairer et al., 2000]. This is a Runge-Kutta (RK) integration of 8^{th} order using RK schemes of order 5 and 3 for error estimation and adaptive step size control and provides dense output. "The performance of this code, compared to methods of lower order, is impressive." [Hairer et al., 2000]

8.2.1 Visualization Environment and Data Model

We use the *VISH* [Benger et al., 2007] visualization shell as our implementation platform. *VISH* supports the concept of a fiber bundles data model [Benger, 2004], which is a hierarchical data model structured in six levels. These levels are called Bundle, Slice (time), Grid, Topology(Skeleton), Representation and Field. Fields store the actual data arrays while the other levels are used similar to a directory structure to organize the data. Datasets such as position, velocity or connectivity information among points are stored in a Field. The collection of Skeletons which hold data Fields in their coordinate Representation is a Grid object. The collection of Grid objects over all time Slices is the Bundle of the dataset.

VISH uses a network structure to separate tasks in atomic entities, called *VISH*-objects, which are connected by input and output connections. A pull model is used updating using a separated control and data flow. Several levels of caching are provided throughout the network updating process [Benger et al., 2009a].

8.2.2 The Integration Module

To compute and visualize the integration geometry the task is split in three parts: definition of the seeding (emitter) geometry, integration based on a data field and rendering. Each task is taken care of by a different module in the *VISH* network.

Here, we present the framework that provides a computational module based on template specializations. Template programming allows to write flexible and reusable code, without performance losses due to late binding or loose coupling. The compiler directly inserts source code of template specializations dependent on their template parameters, which can be highly optimized during compilation. The

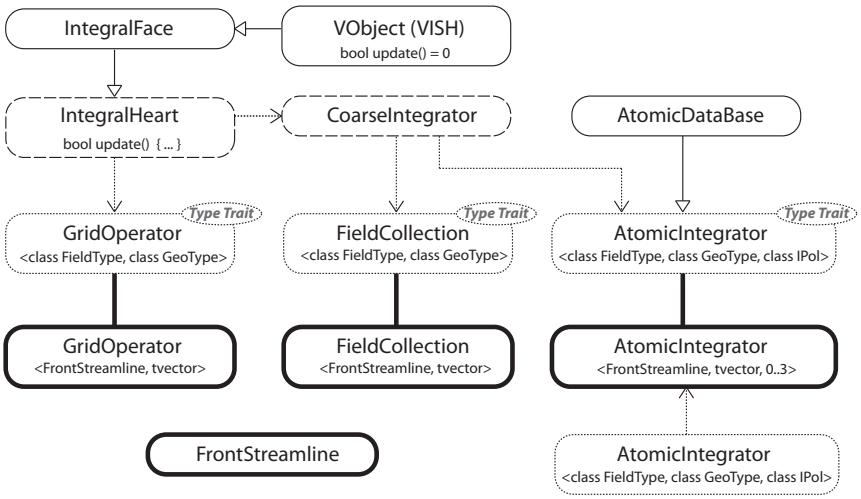


Figure 8.2: Class organization for the integration framework. Classes illustrated in dashed lines are template classes which define an interface but also provide a default implementation. Classes illustrated in dotted lines are template traits which only define an interface by empty functions. They have to be specialized and implemented. Four classes have to be implemented, indicated by the bold outline. The implementation of streamline integration is shown as an example.

overhead of function calls is not relevant in that context since it is removed during compilation. See section (8.3.2) for a comparison of timings runtime measurements between `optimize` versus `debug` compilation. Template programming provides a programming language on its own "executed" at compile time [Veldhuizen, 1995] [Furnish, 1998] [Vandevoorde & Josuttis, 2003].

Fig. 8.2 illustrates the main class relationships for the example of a streamline integration. The four classes outlined in bold have to be customized for that purpose: `GridOperator`, `FieldCollection` and `AtomicIntegrator` have to be specialized and an empty class defining a type has to be introduced. Here, this class is called `Streamline` in Fig. 8.2.

We derive a class called `IntegralFace` from `VObject` and define all the input and output connections necessary for a computation module. Derivation from `VObject` makes a *VISH* network module. `IntegralFace` implements no additional functionality. It provides the following input connections: input initial grid, input base field, interpolation type of the base field (linear, cubic, analytic), integration type (Euler, Dop853), length of integration geometry (or their trajectories), step size and maximum steps. The integration geometry grid is output by the module. From `IntegralFace` we derive the central computation class called

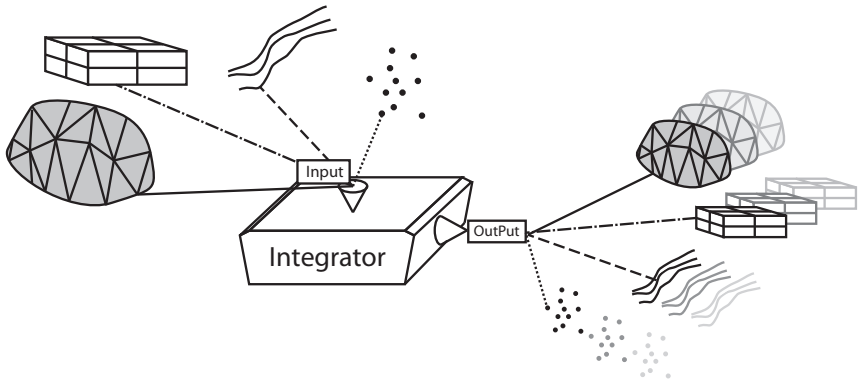


Figure 8.3: Any Grid object can define the seeding geometry and input to and output from the integrator. Examples are a triangular surface, a uniform grid, lines and points. A Grid may also hold data, for example scalar or vector field data. The output Grid depends on the input Grid and is of the same type but with an additional dimension.

`IntegralHeart`:

```
template <typename FieldType, typename IntGeoType>
class IntegralHeart : public IntegralFace
```

It implements the `update()` function used in the network updating process and hosts the main integration loop. Tasks such as retrieving the initial geometry, the data field and the user control parameters and outputting the integration geometry are brought together here. The module handles all different kinds of input grids and data fields by utilizing the fiber bundle library. A input grid can, for example, be a point cloud, a surfaces or a connected curvilinear grid, as illustrated in Fig. 8.3. It is also possible to have additional data such as directional fields stored in the emitter grid, if additional initial conditions besides positions are required. Similarly, the field used in the integration can be of arbitrary type such as scalar, vector or a tensor field specified in any topology or representation.

The two template parameters `FieldType` and `IntGeoType` describe the types of the integration field and the type of the integration geometry. In the streamline example the field type would be a vector field and the integration geometry the empty class `Streamline`. The main computation loop utilized template type trait classes for doing the integration, Fig. 8.4, illustrated by the dotted outlined classes in Fig. 8.2. The traits define empty functions being called from `IntegralHeart`. One additional template parameter is introduced: `int InterpolationType`. It controls the choice of the interpolation scheme used in the integration field.

The main loop uses two template trait classes, see Fig. 8.2: `GridOperator` and `CoarseIntegrator`. The `GridOperator`

```
template <class FieldType, typename IntGeoType>
class GridOperator {...}
```

is responsible for the geometry that is created during the integration process and has to be implemented for a specific type. Several functions may be provided in the template specialization to control the geometry: `prepare()`, `advance()`, `refine()`, `store()` and `finalize()`. The functions `prepare()` and `finalize()` allow initial and final operation on the grid. The other functions are called in each call of the main loop and are responsible for advancing, storing and refining the grid. Using only `Grid` objects as function parameters allows maximum flexibility and power in the functions to modify or create geometry. All topological information is available in a `Grid`, which is necessary in the `refine()` function in case of doing adaptive geometry refinement.

The `CoarseIntegrator` trait provides two functions to the main loop:

```
template <class FieldType, typename IntGeoType, int InterpolType>
class  CoarseIntegrator {...}
```

`advance()` and `extractLocalData()`. The `advance()` function is responsible for the integration in a coarse sense. It itself uses the trait `AtomicIntegrator`

```
template<typename FieldType, typename IntGeoType, int InterpolType>
class AtomicIntegrator {...}
```

which implements the integration on a low level per point basis. The `CoarseIntegrator` advances a collection of points. The default template implementation does a breadth-wise integration by advancing fronts. A depth-wise integration or line-wise integration can be added by providing a different template specialization. The `extractLocalData()` function collects all data besides the vertex data by again calling the according function from the `AtomicIntegrator` over a the collection of vertices which utilizes the `FieldInterpolator` template, a fully implemented template class that returns a linear or cubic interpolation value of a given point in the field. The interpolator also can return an analytic value if a formula for the data field available explicitly.

The `advance()` function of the `CoarseIntegrator` extracts data fields of the current grid object into the so called

```
template <class FieldType, typename IntGeoType>
struct FieldCollection : public MemCore::ReferenceBase
                         <FieldCollection<FieldType,IntGeoType> >
{ ... } .
```

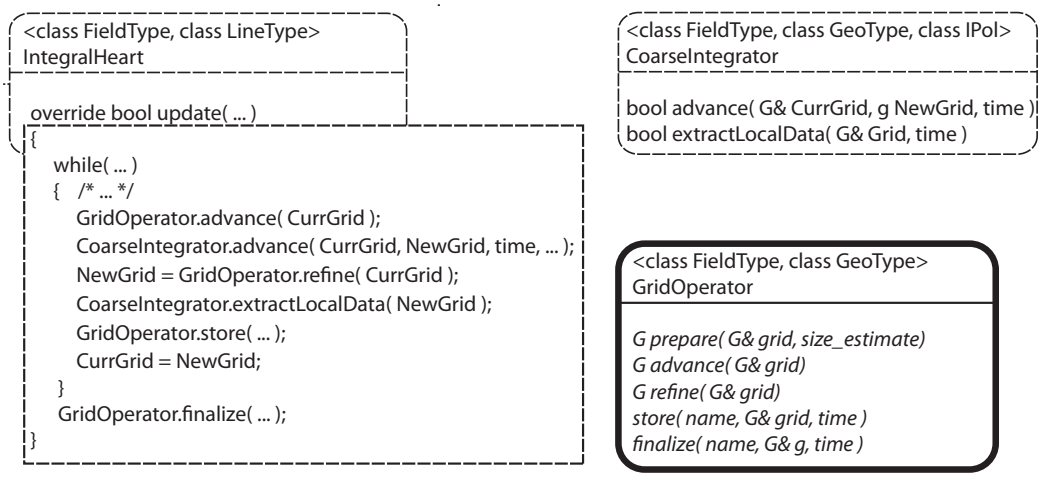


Figure 8.4: More detailed illustration of class functions. All functionality is inserted into the `IntegralHeart`'s `update()` function by the compiler which uses the template trait functions of `GridOperator` and `CoarseIntegrator`. Reference pointers are used when pointers are required. `G` is short for `RefPtr<Grid>`.

The field collection provides data necessary for integration on an array basis used by the functions of the `AtomicIntegrator`. Thus, the atomic integration operation need not to extract this data from the grid in every integration step. `AtomicIntegrator` provides doEuler() and doDop853() functions to process every points on the integration coarse.

The following subsections describe the implementation of four different integration geometries using the presented framework.

8.2.3 Streamline Implementation

To implement the streamline integration several classes are gathered in a separate `cpp` file. An empty class `class FrontStreamline{};` is defined and the `GridOperator` specialized:

```
template<> class GridOperator<tvector, FrontStreamline> {...}
```

where `tvector` is the type of the vector field. Its `prepare()` member function retrieves the emitter grid and prepares a new integration geometry grid by copying the vertices. The geometry will later be stored as a set of lines in the bundle of the vector field. It also estimates the size of the data being computed and reserves memory. The `advance()` just passes the given grid through without any

operation. The `refine()` function also passes the grid through. Refinement will by implemented in future. The `store()` function is an empty function. Nothing is stored per time-step. Finally, the `finalize()` function stores the filled data fields at the given time step into the bundle.

The `FieldSelection` specialization extracts `std::vector`s from the integration geometry grid and provides them to the `AtomicIntegrator` partial specialization:

```
template<int InterpolType>
class AtomicIntegrator<tvector, FrontStreamline, InterpolType>
: public AtomicDataBase {...};
```

Here, the `doEuler()` function is implemented which uses an index to access the correct vertex and direction. These are retrieved from the `Field- Collection` provided by the `CoarseIntegrator`. It computes and pushes the next vertex position and line connection into the `FieldCollection`. The `extractLocalData()` function extracts the interpolated vector field data for the vertex positions previously computed by the integration function by utilizing the `FieldInterpolator` template.

Finally, a *VISH* network module is provided by a template instantiation:

```
typedef IntegralHeart<tvector,FrontStreamline> FrontStreamlines;
```

Implementing streamline computation requires 350 lines of source code.

8.2.4 Pathlines Implementation

The pathline integration again requires a type `class FrontPathline{};` and a specialization of

```
template<> class GridOperator<tvector, FrontPathline> { ... };
```

The representation of a pathline differs from the representation of a streamline in the fiber data bundle. For a pathline a vertex is stored as grid in different time slices. The `prepare()` function also creates a new grid object and copies the vertex data from the emitter grid. However, the `advance()` function now creates a new grid object for each step of the integration front. The `refinement()` function is to be implemented in future and just passes the grid though. The `store()` function inserts the computed grid into the bundle for each integration front step. Here, the `finalize()` function is an empty implementation. The `FieldSelection` again extracts the array data from the grid object and provides array data.

The specialization of

```
template<int InterpolType>
class AtomicIntegrator<tvector, FrontPathline, InterpolType>
: public AtomicDataBase {...};
```

implements the `doEuler()` function, using a provided time step value. It uses direct array index access instead of using push backs (in the streamline case). The `extractLocalData()` function interpolates the vector field and stores directions at each vertex.

A *VISH* module is provided by the template instantiation:

```
typedef IntegralHeart<tvector,FrontPathline> FrontPathlines;
```

Pathline computation requires 250 lines of source code.

8.2.5 Material-Line Implementation

The pathline module can be reused for the material-line implementation. The fiber bundle data stores topology information inside a Grid: A so called relative Representation on the vertices in a Skeleton. The pathline module is extended, copying this additional information in the `GridOperators` `prepare()` and `advance()` functions. This was implemented for an arbitrary number of additional Skeletons on the vertices using a Skeleton iterator. 50 additional lines are required in the framework and 4 lines for the pathline implementation.

8.2.6 Time Surfaces

The computation of the time surface did not need any additional development because the Skeleton iterator, implemented for material-lines, already copies the topological surface connectivity data.

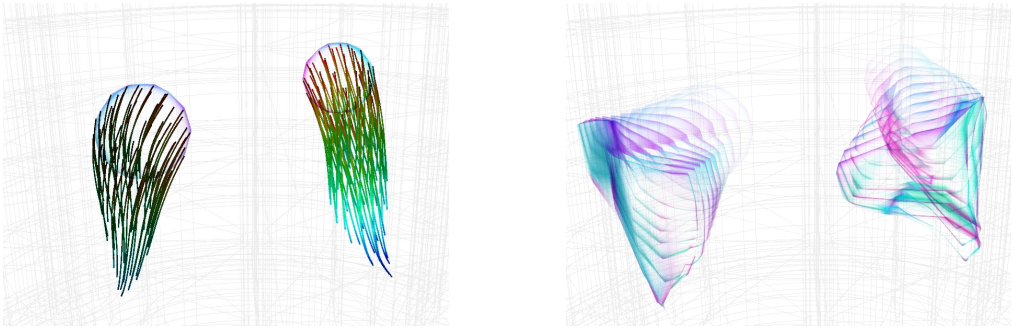


Figure 8.5: Images showing the streamlines [left] and the evolving time surfaces [right], with a two sphere geometry as a seeding grid.

8.3 Results

8.3.1 CFD Visualization of a Stirred Tank

Having the two computing modules implemented for streamline and pathline integration and having extended the modules providing the seeding geometries we are able to visualize streamlines, pathlines, material-lines and time surfaces in a uniform test grid and in a 2088 multi-block curvilinear dataset stemming from a CFD simulation of a stirred tank. The curvilinear grid is comprised of 3.1 million cells in total, with flow variables such as velocity and pressure measured at the cell vertices [Benger et al., 2009c]. The integration module currently implemented is explicit Euler. The figures show the results of the integration in the curvilinear dataset. The computation was done for 50 time steps using SVN revision 2557 of *VISH*. The material-lines, time surfaces and pathlines in Fig. 8.6 and 8.7 are rendered for every 5^{th} time step. They are fading linearly in time, highlighting the current state.

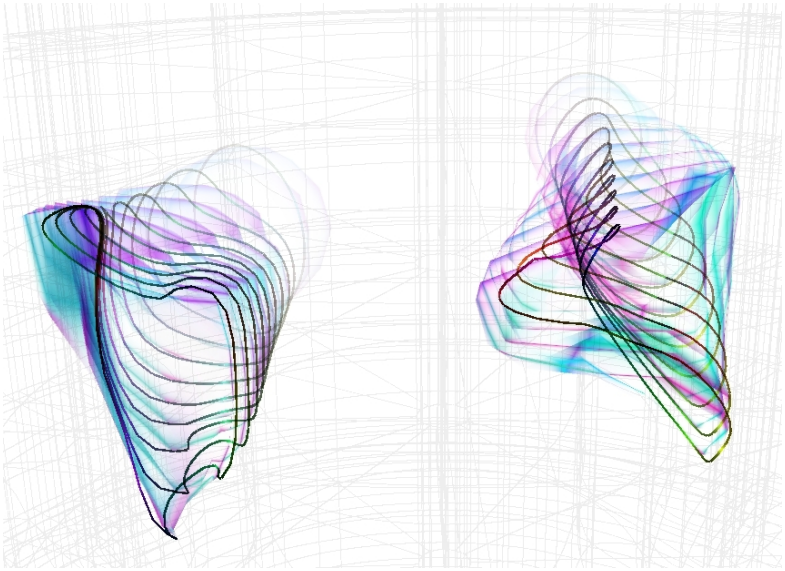


Figure 8.6: Image showing corresponding time surfaces overlapped on material lines [right], both computed for 50 time steps.

8.3.2 Time Measurements

We did measurements of the computation time comparing our old implementations and the new ones in a simple uniform grid based dataset and the stirred tank data set. For testing we used a machine equipped with a six core Intel Xeon

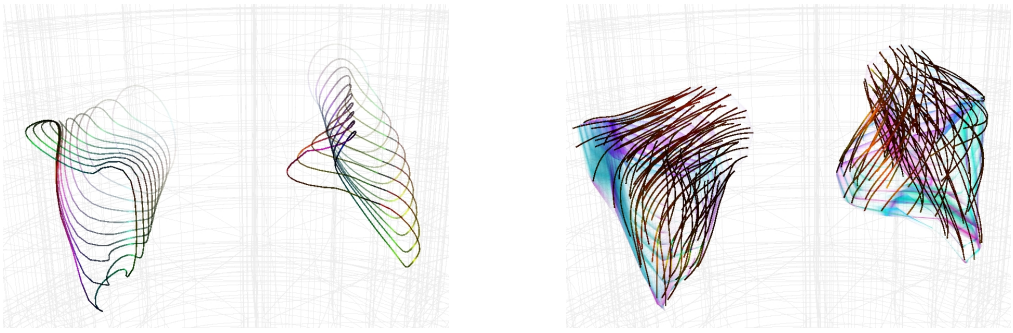


Figure 8.7: Images showing the material lines [left] and time surfaces with pathlines overlapped [right], all computed for 50 time steps.

W3680@3.33GHz, 12MB L3, 6.4GT/s with 6GB of 1333MHz DDR3 SDRAM and a NVIDIA Quadro FX 3800 1GB running gcc version 4.4.4 20100630 (Red Hat 4.4.4-10).

Imple-mentaion	**Data-set**	**Comp-ilation**	**Steps** #	**Step-Time** [msec]	**N.-Time** [%]	**Spd-Up** [-]	**Spd-Up** [-]
stream old	uni.	debug	2704	0.19	*100*		
stream old	curvi.	debug	6739	3.10	100		
stream old	uni.	opt.	2704	0.03	***16***	*6.3*	
stream old	curvi.	opt.	6739	1.40	45	2.2	
stream new	uni.	debug	2600	0.19	*100*		1.0
stream new	curvi.	debug	6600	5.40	174		**0.6**
stream new	uni.	opt.	2600	0.03	***16***	*6.3*	1.0
stream new	curvi.	opt.	6600	2.60	84	2.1	**0.5**
path old	uni.	debug	2600	0.20	*105*		
path old	curvi.	debug	6732	429.88	13867		
path old	uni.	opt.	2600	0.03	***16***	*6.7*	
path old	curvi.	opt.	6732.21	224.00	7226	1.9	
path new	uni.	debug	2600	0.21	*111*		1.0
path new	curvi.	debug	6600	5.61	181		**76.6**
path new	uni.	opt.	2600	0.03	***15***	*7.0*	1.0
path new	curvi.	opt.	6600	2.61	84	2.1	**85.8**

The table gathers: Old and new stream and pathline using debug and optimized compilation, number of integration steps, time per integration step, normalized-time with respect to the old uniform and curvilinear grid streamline computation, speedup by optimized compilation mode and speedup by the new implementation.

When comparing the timings of the old and the new integration in the uniform grid the measurements show no difference. The introduced overhead of the

framework does not result in a longer computation time. In the curvilinear case of streamlines the old computation is faster by a factor of about two. This has to be investigated. However, the new curvilinear pathlines benefit a speedup of about factor of 80 by making faster interpolation and point search algorithms available.

8.4 Conclusion

While the earlier version of integration modules were implemented independent of each other with redundant computation code and time, we successfully designed and implemented a framework based on template specializations that provides a common computational module for different integral geometries. We introduced the visualization of material lines with minimal programming effort: 350 lines for streamlines, 250 lines for pathlines, 54 lines for material lines and none for time surfaces.

8.5 Future Work

We will adapt our existing three and four dimensional geodesic tensor field integration code to the framework, enable DOP853 integration, introduce grid refinement during integration, introduce a module to extract a path-surface from computed material lines and work on a thread-based parallelization on the CPU using OpenMP [OpenMP Architecture Review Board, 2010].

Acknowledgments

We thank the VISH development team, among them Georg Ritter, Science and Technology Research Institute at the University of Hertfordshire, and Hans-Peter Bischof, Rochester Institute of Technology, for their support. This research employed resources of the Center for Computation & Technology at Louisiana State University and was supported by the projects “Modeling and Visualizing the Effect of Severe Storms on Oil Spill Trajectories with the Cactus Framework” and “Visualization of Oil Spill Related Data” awarded by the LONI Network Deployment & Operation [The LONI Institute, 2010].

Bibliography

[Benger, 2004] Benger, W. (2004). *Visualization of General Relativistic Tensor Fields via a Fiber Bundle Data Model*. PhD thesis, FU Berlin.

[Benger et al., 2007] Benger, W., Ritter, G., & Heinzl, R. (2007). The Concepts of VISH. In 4^{th} *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007* (pp. 26–39).: Berlin, Lehmanns Media-LOB.de.

[Benger et al., 2009a] Benger, W., Ritter, G., Ritter, M., & Schoor, W. (2009a). Beyond the visualization pipeline. In 5^{th} *High-End Visualization Workshop, Baton Rouge, Louisiana, March 18th - 21st, 2009*: Berlin, Lehmanns Media-LOB.de.

[Benger et al., 2009b] Benger, W., Ritter, G., Su, S., Nikitopoulos, D. E., Walker, E., Acharya, S., Roy, S., Harhad, F., & Kapferer, W. (2009b). Doppler speckles - a multi-purpose vectorfield visualization technique for arbitrary meshes. In *CGVR'09 - The 2009 International Conference on Computer Graphics and Virtual Reality*.

[Benger et al., 2009c] Benger, W., Ritter, M., Acharya, S., Roy, S., & Jijao, F. (2009c). Fiberbundle-based visualization of a stir tank fluid. In 17^{th} *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (pp. 117–124). URL: `http://wscg.zcu.cz/WSCG2009/Papers_2009/!_WSCG2009_Short_final.zip`.

[Bohara et al., 2010] Bohara, B., Harhad, F., Benger, W., Brener, N., Iyengar, S., Ritter, M., Liu, K., Ullmer, B., Shetty, N., Natesan, V., Cruz-Neira, C., Acharya, S., Roy, S., & Karki, B. (2010). Evolving time surfaces in a virtual stirred tank. In V. Skala (Ed.), *18th International Conference on Computer Graphics, Visualization and Computer Vision'2010*.

[Deuflhard & Bornemann, 2002] Deuflhard, P. & Bornemann, F. (2002). *Scientific Computing with Ordinary Differential Equations*. Springer Verlag, New York.

[Furnish, 1998] Furnish, G. (1998). Container-Free Numerical Algorithms in C++. *Computers in Physics*, 12(3).

[Hairer et al., 2000] Hairer, E., Norsett, S. P., & Wanner, G. (2000). *Solving Differential Equations I*. Springer-Verlag Berlin Heidelberg.

[Krishnan et al., 2009] Krishnan, H., Garth, C., & Joy, K. I. (2009). Time and streak surfaces for flow visualization in large time-varying data sets. *Proc. IEEE Visualization '09*.

[McLoughlin et al., 2009] McLoughlin, T., Laramee, R. S., & Zhang, E. (2009). Easy integral surfaces: a fast, quad-based stream and path surface algorithm. In *Proceedings of the 2009 Computer Graphics International Conference*, CGI '09 (pp. 73–82). New York, NY, USA: ACM. URL: `http://doi.acm.org/10.1145/1629739.1629748`.

[OpenMP Architecture Review Board, 2010] OpenMP Architecture Review Board (2010). OpenMP. URL: `http://openmp.org/wp/`.

[Ritter, 2010] Ritter, M. (2010). Computing Geodesics in Numerical Space Times. Master's thesis, Institute of Computer Science, University of Innsbruck. `http://www.fiberbundle.net/papers/da_geodesics_print.pdf`.

[Ritter & Benger, 2010] Ritter, M. & Benger, W. (2010). Visualizing coordinate acceleration and christoffel symbols. *IADIS Computer Graphics, Visualization, Computer Vision and Image Processing 2010 (CGVCVIP 2010)*.

[Stalling, 1998] Stalling, D. (1998). *Fast Texture-Based Algorithms for Vector Field Visualization*. PhD thesis, Free University Berlin.

[The LONI Institute, 2010] The LONI Institute (2010). LONI. URL: `http://www.loni.org/`.

[Vandevoorde & Josuttis, 2003] Vandevoorde, D. & Josuttis, N. M. (2003). *C++ Templates - The Complete Guide*. Addison Wesley.

[Veldhuizen, 1995] Veldhuizen, T. L. (1995). Using C++ template metaprograms. *C++ Report*, 7(4), 36–43. Reprinted in C++ Gems, ed. Stanley Lippman. URL: `http://extreme.indiana.edu/~tveldhui/papers/`.

[Weinkauf & Theisel, 2010] Weinkauf, T. & Theisel, H. (2010). Streak lines as tangent curves of a derived vector field. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization 2010)*, 16(6), 1225–1234. Received the Vis 2010 Best Paper Award. URL: `http://tinoweinkauf.net/`.

Article 9

Improved Visualisations of 3D Volumetric Data through Pointwise Phong Shading Based on Normal Mapping

Josef Stöckl[1], Dominik Steinhauser[1], Markus Haider[1], Tobias Riser[1]

[1]Institute of Astro- and Particle Physics University of Innsbruck
josef.stoeckl@uibk.ac.at

Numerical simulations have become increasingly important in almost any field of physics throughout the last decades, since many physical systems are governed by coupled partial differential equations, for which no analytic solution exists. The analysis and visualisation of the resulting amounts of 3D data is one of the central problems in modern science. Although several commonly used visualisation packages can work with volumetric data, many of them only provide limited capabilities in producing a suitable depth perception and could be improved using advanced methods. One such method, pointwise Phong shading based on normal mapping, and its advantages are described in detail in this paper.

[1]All authors contributed equally to this work.

9.1 Introduction

The continuous performance gain of computers within the last decades made numerical simulations an ideal tool to study complex physical systems which are often described by coupled partial differential equations for which no analytical solutions exist. Therefore numerical simulations now play an important part in almost any field of physics.

Our working group in Innsbruck, for example, is doing simulations on the formation of galaxy clusters with a special emphasis on the hydrodynamics of the intra-cluster gas ([Kapferer et al., 2007]). Our simulations provide us with three-dimensional arrays for quantities like the temperature, density and metallicity of the cluster gas. In order to understand the physical content of the simulations, we have to visualise this data (see e.g. [Kapferer & Riser, 2008]). There are several commercial (e.g. Amira[1]) as well as free (e.g. Paraview [2], Visit [3]) programs available for visualising volumetric data. Approximately speaking, the visualisation of 3D data in these programs works as follows: Every grid cell of the 3D volume gets assigned a colour and alpha value based on a colour map corresponding to the value of the quantity which should be visualised. Then, through a projection via some specified perspective, a 2D image is generated out of the 3D data. In order for all the grid cells to contribute to the image, transparency has to be applied in the projection. However, sometimes it is difficult to see the 3-dimensional structure in the images resulting from these projections, as projections are ambiguous. Many programs try to solve this ambiguity by letting the user control the perspective in real time, as viewing the data from different angles provides us with the necessary information about the 3D structure. There is an additional technique to improve the presentation of 3D data: Shading based on normal mapping. This technique is well known, but as it is absent in many of the commonly used packages, we want to draw attention to it.

In section 9.2 we will provide some background information about lighting and shading, and in section 9.3 we will describe the method as implemented in our visualisation program together with some comparisons of renderings with shading on and off.

9.2 Theory

In this section some theories about three-dimensional visualisation techniques, used in this work, are described. First, lighting and shading, necessary for the per-

[1] `http://www.amira.com`
[2] `http://www.paraview.org`
[3] `https://wci.llnl.gov/codes/visit/`

ception of three-dimensional structure are introduced. Furthermore, direct volume rendering is depicted, which is used to display a two-dimensional projection of a three-dimensional discrete data set.

9.2.1 Lighting and Shading

Whether in real life or in computer graphics, a three-dimensional scene must be illuminated. Thereby the scene gets illuminated by one or more light sources which are characterised by a collection of properties like position, pointing, colour and intensity of the light. Furthermore, different types of light sources and varying parameters can be introduced, as for example sources with light-direction-dependent colours, uniformly radiating point sources or directed lamps and spotlights. The light sources and their properties combined result in a lighting model. Generally, light sources can be divided into *local* and *infinite* or *remote light sources*. Local light sources are mainly used for particular effects, whereas infinite light sources illuminate all objects the same way since the light rays are parallel and pointing in the same direction and hence less expensive in computational cost.

Beside the light sources, also the ambient light is of great importance which does not come directly from a light source but is arriving at a surface by being reflected at other objects in the scene. Since the computation of the amount and direction of reflected light is difficult and time consuming, different simplifications are adopted often. A constant amount of ambient light can be added to every object in the scene whenever its illumination has to be found ([Glassner, 1989], [Angel, 1990]).

Once all the light sources, the material properties and the geometry of the considered object have been defined, reflection and -in case of partly transparent objects- transmission of the light can be calculated. The process of determining the properties of the light leaving the surface is called *shading*. Shading is crucial to depict the depth perception of three-dimensional objects. In principle, it is possible to shade any surface by processing each single, visible point of a scene. Knowing the illumination model at that point, the shading can be calculated by just calculating the surface normal at the considered point, since irradiance and reflection can easily be calculated.

Unfortunately, this method is computationally too expensive. Using shading on polygons, the simplest shading model that can be adopted is *constant shading*, where for each polygon just one intensity value is calculated and kept throughout the whole polygon. This approach is not usable for surfaces that are an approximation to a curved surface and requires both the light source and the viewer to be at infinite distance. Obviously, not very realistic results can be obtained, therefore more sophisticated methods have been introduced as *Phong shading*, also used in this work, described in the next section. Moreover, beside of varying the bright-

ness according to the shading model, also the colour itself can be changed. This technique is called *tone shading* ([Ebert & Rheingans, 2000]).
On the other hand, pixel based shading methods are nowadays commonly used, especially in computer games. Worth mentioning is normal mapping or Dot3 bump mapping. As in ordinary bump mapping, textures of a higher refined model are used. The normal vector information for calculating the diffuse lighting at the surface at each point is encoded in a 3-channel (a channel for each dimension) bitmap and combined with the bump map. This gives the illusion that the model is more detailed than it actually is.

9.2.2 Phong shading

Phong shading was introduced in 1975 by Bui Tuong Phong ([Phong, 1975]) and is also known as normal-vector interpolation shading ([Foley et al., 1994]). This method offers significant advantages over previously used ones, most notably the perception of a smooth surface and the avoidance of visual artifacts at the edges of polygons (see [Foley et al., 1994] for a more detailed discussion). As the name already implies, this shading technique is based on the interpolation of normal vectors: Starting from the two normal vectors on the span of a polygon on a scan line, the normal vectors in between are interpolated. These two starting vectors are themselves interpolated from the vertex normal vectors which again can just be determined as in Gouraud shading. The interpolation of normal vectors is done as follows:

$$N_t = t\,N_1 + (1-t)\,N_0 \tag{9.1}$$

with N_0 and N_1 being the normal vectors at two vertices and $0 \leq t \leq 1$. After being normalised, the normal vectors at each pixel on the scan line are used to calculate the new intensity using the lighting models to calculate the shading. It is assumed that the light emitted by an object and getting to the eye of an observer comes from combined diffuse and specular reflection of the irradiated light. Hence, the shading at an object's point p can be calculated as

$$S_p = C_p\left[\cos(i)\,(1-d) + d\right] + W(i)\cos^n(s) \tag{9.2}$$

with i and s being the incident angle and the angle between direction of light and line of sight, C_p a reflection coefficient, $W(i)$ the ratio of specular reflected and incident light depending on the incident angle, d the diffuse reflection coefficient and n models the specular reflection. The specular reflection term is the reason why this shading method provides a good treatment of highlights and is therefore well suited for curved surfaces.

9.2.3 Direct volume rendering

Direct volume rendering (DVR) is a method used to obtain a two-dimensional image out of a three-dimensional scalar field or data set. Unlike to slicing three-dimensional data or plotting iso-surfaces, DVR has the advantage that all the data can potentially contribute to the output image.
First, the three-dimensional data can be represented by voxels. Voxels (volume picture elements) are the three-dimensional analogue to pixels. In principle, there are two methods describing voxels. The first is to define a cubic cell and assign a single colour and possibly a transparency to it. Another possibility is to define data points at the corner of cubic cells. Then the colour and transparency values inside the cells are calculated by interpolation.
The transparency of the voxels ensures, that all these cells can contribute to the final image. Therefore, a visualisation of the whole three-dimensional data set is possible. The two-dimensional image is consequently a projection of the complete data set. A possible method for doing this is ray-casting, whereas a ray outgoing from each pixel of the final two-dimensional image into the object space is traced. After calculating the colour value along the ray, a value to the pixel in the two-dimensional image is assigned.
Usually, images obtained from DVR are blurry and fuzzy because many points in the data set are considered which can also lead to ambiguity in the depth perception. This problem can be solved by providing interactivity, if the volume and the viewpoint can be changed, a much better feeling for the data set can be obtained.
Also Amira uses DVR in its *Voltex* module. A colour map including alpha values for transparency is used to calculate the amount of emitted and absorbed light of each voxel. Then the projection of the voxels in the data set is computed by drawing slices from back to front and composing them. The slices are textured polygons, whereas the texture is computed by using the data and colour map. An example is shown in figure 9.1.

This method is also denominated *texture based volume rendering* and can also be applied to 2D textures, whereas the voxel data is composed into three stacks of 2D texture slices. The stack which is most parallel to the viewing axis is going to be rendered. As in ([Westermann & Sevenich, 2001]), several approaches can be used to reduce computation time and memory consumption using this method as for example ray-traversal, omitting homogeneous and empty regions.

Furthermore, instead of one-dimensional transfer functions, which maps the scalar value of each cell of the data set to colour, opacity etc., multi-dimensional transfer functions which take multiple measures (instead of the one scalar) into account, can be used. However, the design of such functions is not straightforward. Often, a discrete set of values is stored in lookup tables for easy and fast access.

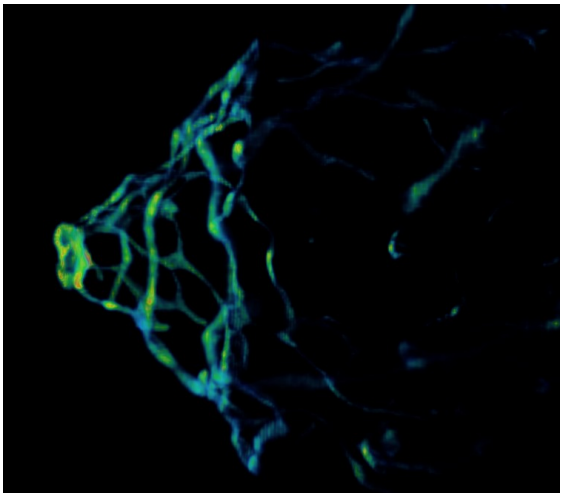


Figure 9.1: In this image, the gaseous component of a ram pressure stripped galaxy in a side-on view is shown using the Voltex module of Amira. The transparency allows the visualisation also of the gaseous filaments rearmost in the scene.

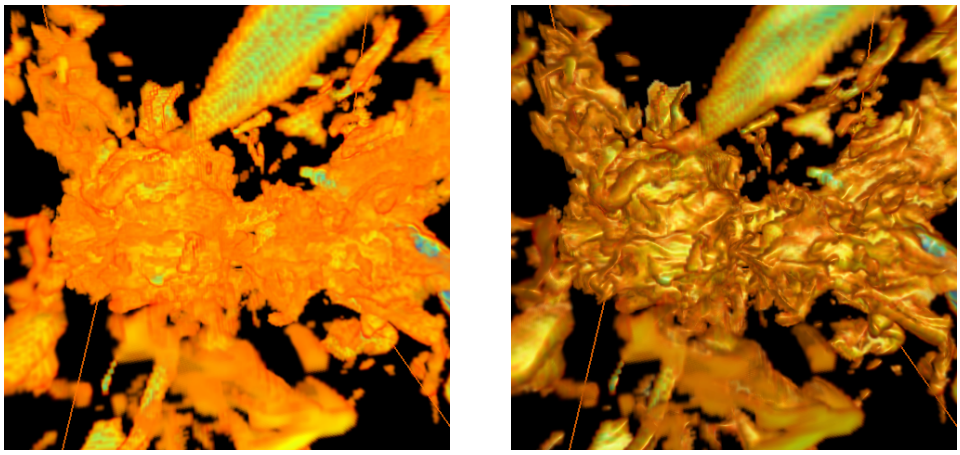
Anyway, for multivariate data, these tables can become immensely large. Therefore, as in ([Kniss et al., 2003]), a sum of Gaussian transfer functions are used, because their integral can easily be calculated on the GPU, to avoid the use of large lookup tables.

9.3 Improving the 3D experience

One of the main issues in the visualisation of volumetric data is the actual usage of the third dimension. In principle there are two ways of generating a three-dimensional perception: (a) directly by providing suitable images for both eyes, or (b) indirectly by tricking the brain into self-constructing a 3D impression from a 2D image.

The first method proves still to be costly and complicated, although recent developments coming from the gaming and movie industries have brought three-dimensional viewing considerably more into the mainstream market. Still the used technologies suffer from problems (e.g. people being unable to see the 3D effect, headaches and dizziness after longer viewing periods) and severe limitations (e.g. viewing angle, uncomfortable glasses, brightness of the image) and can at the moment only be considered as moderately usable.

Indirect methods of inducing depth perception have been known for a long time and come in a number of varieties: hidden objects, simulated shadows, freeview pictures, autostereograms, and many more. Most of these methods either require



(a) no shading (b) Phong shading

Figure 9.2: An example of the metal distribution in the intra-cluster medium (ICM) of a cluster of galaxies (a) without and with (b) Phong shading (plus normal mapping). In (a) the depth information of the data slices is lost and in regions where similar data values overlay each other, the position of the data points can hardly be determined by the viewer. In contrast the shading in (b) provides a natural depth perception.

a special kind of viewing (e.g. parallel staring into infinity or cross-eyed focusing) or use brightness information (shadows, lighting) to stimulate the brain into constructing a 3D perception. Since these methods do not require any technology, they are often found in printed media, but are also widely used in the computer and gaming industries.

9.3.1 Shading and depth perception

The fact that the brightness information of a scene causes a natural three-dimensional impression enables us to improve the volume rendering of datasets without the need for special and expensive hardware. As outlined in section 9.2.2, Phong shading provides the fastest nearly realistic way to calculate the lighting of curved surfaces. Although more sophisticated and physically correct shading techniques like the Cook-Torrance ([Cook & Torrance, 1982]) or the He-Sillion-Torrance-Greenberg ([He et al., 1991]) shading exist, they come at the price of much higher computational costs and yield no improvement in the visualisation of volumetric datasets. Since these physics based shading models only improve on the correct representation of surface materials, the use of a phenomenologically realistic model like the Phong shading is sufficient to achieve an accurate depth perception.

How easy and useful it is to provide some depth perception from a 2D image can

be seen in Fig. 9.2. While the iso-contour without shading (Fig. 9.2a) doesn't allow for any three-dimensional image to form, the shaded version (Fig. 9.2b) naturally allows our mind to construct the scene in 3D. The image can therefore transport considerably more information about the structure and distribution of the data.

In the following sections we will describe a method of using Phong shading together with volumetric normal mapping to provide a natural depth perception when visualising volumetric data. The gains of this method and a few examples will be shown and then discussed in section 9.4.

9.3.2 The shading procedure in detail

The method presented here uses pointwise Phong shading and volumetric normal mapping to create an impression of optical depth when used to visualise volumetric data. The whole visualisation procedure works in a four-step process: 1. calculating the colour texture and normal vector for each data point, 2. dividing the domain into slices which are perpendicular to the line of sight, 3. applying normal mapping and shading on the slices, 4. summing the slices up.
The following sections will detail the individual steps.

Calculating the colour texture and normal vectors

Let us consider volumetric data consisting of a rectangular cuboid with $n_x \times n_y \times n_z$ data points $D(x, y, z)$, where $1 \leq x \leq n_x$, $1 \leq y \leq n_y$, and $1 \leq z \leq n_z$. Together with a colour map $C\,(v \in [0; 1])$ each point can be assigned a colour and alpha value, depending on the colour space used and any selected data threshold values $D_{\min}$ and $D_{\max}$. Mathematically this assignment can be represented by a texture transfer function Γ:

$$\Gamma(x, y, z) = C\left(\frac{\tilde{D}(x, y, z) - D_{\min}}{D_{\max} - D_{\min}}\right), \tag{9.3}$$

where

$$\tilde{D}(x, y, z) = \begin{cases} D_{\min} & \text{if } D(x, y, z) < D_{\min}, \\ D_{\max} & \text{if } D(x, y, z) > D_{\max}, \\ D(x, y, z) & \text{otherwise.} \end{cases} \tag{9.4}$$

For the normal vectors, we have to calculate the local data gradient at each data point. This can be done in various more or less sophisticated ways: The simplest and fastest approach is the central difference operator $\vec{\nabla}_{\mathrm{CD}}$ which can be defined by

$$\vec{\nabla}_{\mathrm{CD}} D(x, y, z) = \begin{pmatrix} D(x+1, y, z) - D(x-1, y, z) \\ D(x, y+1, z) - D(x, y-1, z) \\ D(x, y, z+1) - D(x, y, z-1) \end{pmatrix}. \tag{9.5}$$

A more robust and accurate method would be the Sobel operator $\vec{\nabla}_{\mathrm{S}}$, which can be expressed by

$$\vec{\nabla}_{\mathrm{S}} D(x,y,z) = \sum_{i,j,k\in\{-1,0,1\}} \begin{pmatrix} S'(i)S(j)S(k)D(x+i,y+j,z+k) \\ S(i)S'(j)S(k)D(x+i,y+j,z+k) \\ S(i)S(j)S'(k)D(x+i,y+j,z+k) \end{pmatrix}, \tag{9.6}$$

where $S(m) = \begin{cases} 1 & \text{if } m=-1, \\ 2 & \text{if } m=0, \\ 1 & \text{if } m=1, \end{cases}$ and $S'(m) = \begin{cases} 1 & \text{if } m=-1, \\ 0 & \text{if } m=0, \\ -1 & \text{if } m=1, \end{cases}$ are defined for $m \in \{-1,0,1\}$. The Sobel operator is especially well suited for noisy data or steep and rapidly changing gradients, but comes at the price of being more than a factor 4 slower than the central difference. In practice the quality and noisiness of the data determines which gradient operator should be used. This often results in trying different operators and comparing the visual result.

Regardless of the operator used to determine the gradient vectors, we can now construct an array of normal vectors $\vec{N}(x,y,z)$:

$$\vec{N}(x,y,z) = \begin{cases} \vec{\nabla} D(x,y,z) & \text{if } \vec{s}\cdot\vec{\nabla}D(x,y,z) < 0, \\ -\vec{\nabla} D(x,y,z) & \text{otherwise,} \end{cases} \tag{9.7}$$

which will be used for the Phong shading and normal mapping. The condition $\vec{s}\cdot\vec{\nabla}D(x,y,z) < 0$ makes sure that the normal vectors are pointing towards the camera, since we have defined the line of sight $\vec{s}$ as pointing from the camera towards the data cube.

Dividing the domain into slices

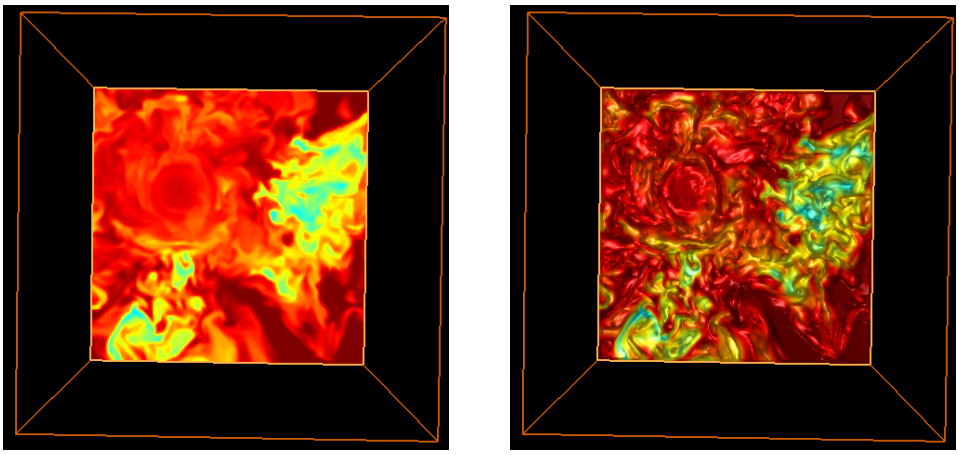
To calculate the volumetric view, the domain has now to be divided into a selectable number n_{sl} of slices which are perpendicular to the line of sight $\vec{s}$. These slices have to fulfill the plane equation

$$\vec{s}\cdot\vec{X} = \lambda_i, \tag{9.8}$$

where $\vec{X} = (\tilde{x},\tilde{y},\tilde{z})$ is a non-discrete positional vector in the data cuboid and $1 < i < n_{\mathrm{sl}}$ denotes the i-th slice whose position is determined by λ_i. If $\lambda_{\max}$ and $\lambda_{\min}$ are the maximum and minimum values of $\vec{s}\cdot\vec{X}$ found at the corner coordinates of the data cuboid, then λ_i can be calculated:

$$\lambda_i = \lambda_{\max} - \left(i - \tfrac{1}{2}\right) * \Delta\lambda, \tag{9.9}$$

where $\Delta\lambda = \dfrac{\lambda_{\max} - \lambda_{\min}}{n_{\mathrm{sl}}}$ is the distance between two slices.



(a) without lighing (b) with lighting

Figure 9.3: A slice of data after being texturised (a) and after Phong shading and normal mapping has been applied (b). It is easily visible that the slice immediately gains depth information and also gives an impression of motion where the gradients are steep.

Applying texture, normal mapping and Phong shading

Since the slices now cross the domain at arbitrary and non-discrete positions $\vec{X}$, the texture and normal vectors in the slice layer have to be calculated by interpolation from the surrounding discrete data points. Using trilinear interpolation each point $\vec{X} = (\tilde{x}, \tilde{y}, \tilde{z})$ in the slice i is assigned a texture Γ_i and a normal vector $\vec{N}_i$:

$$\Gamma_i(\vec{X}) = \sum_{i,j,k \in \{0,1\}} \omega_i(\tilde{x})\omega_j(\tilde{y})\omega_k(\tilde{z})\, \Gamma(\lfloor \tilde{x} \rfloor + i, \lfloor \tilde{y} \rfloor + j, \lfloor \tilde{z} \rfloor + k), \tag{9.10}$$

$$\vec{N}_i(\vec{X}) = \sum_{i,j,k \in \{0,1\}} \omega_i(\tilde{x})\omega_j(\tilde{y})\omega_k(\tilde{z})\, \vec{N}(\lfloor \tilde{x} \rfloor + i, \lfloor \tilde{y} \rfloor + j, \lfloor \tilde{z} \rfloor + k), \tag{9.11}$$

where $\omega_n(\delta) = \begin{cases} \lceil \delta \rceil - \delta & \text{if } n = 0, \\ \delta - \lfloor \delta \rfloor & \text{if } n = 1. \end{cases}$

Although these operations as well as the texture Γ and the normal vectors $\vec{N}$ in section 9.3.2 have been depicted on a regular grid, they only have to be modified slightly to work with unstructured grids. In this case the floor $\lfloor\rfloor$ and ceiling $\lceil\rceil$ functions have to be replaced with proper mapping functions that return the nearest lower and upper grid point, and one has to take care that the summation over i, j, k alternates between these function mappings.

Since modern graphics cards and the corresponding APIs (*Application Programming Interfaces*, e.g. OpenGL) already implement trilinear filtering for tex-

tures and vectors (which are internally treated as 3D-textures), these steps can be performed in the hardware at very high performance.

Now the calculated texture Γ_i is applied to each slice (see Fig. 9.3a for an example) and then the normal mapping and Phong shading is applied using the normal vectors $\vec{N}_i$ (see Fig. 9.3b). Also these operations and the actual rendering of the slices including diffuse, specular and ambient lighting is done in hardware. The resulting contribution of each slice is stored as an image $P_i(\xi, \upsilon)$, where ξ and υ are in screen coordinates.

Creating the final volumetric view

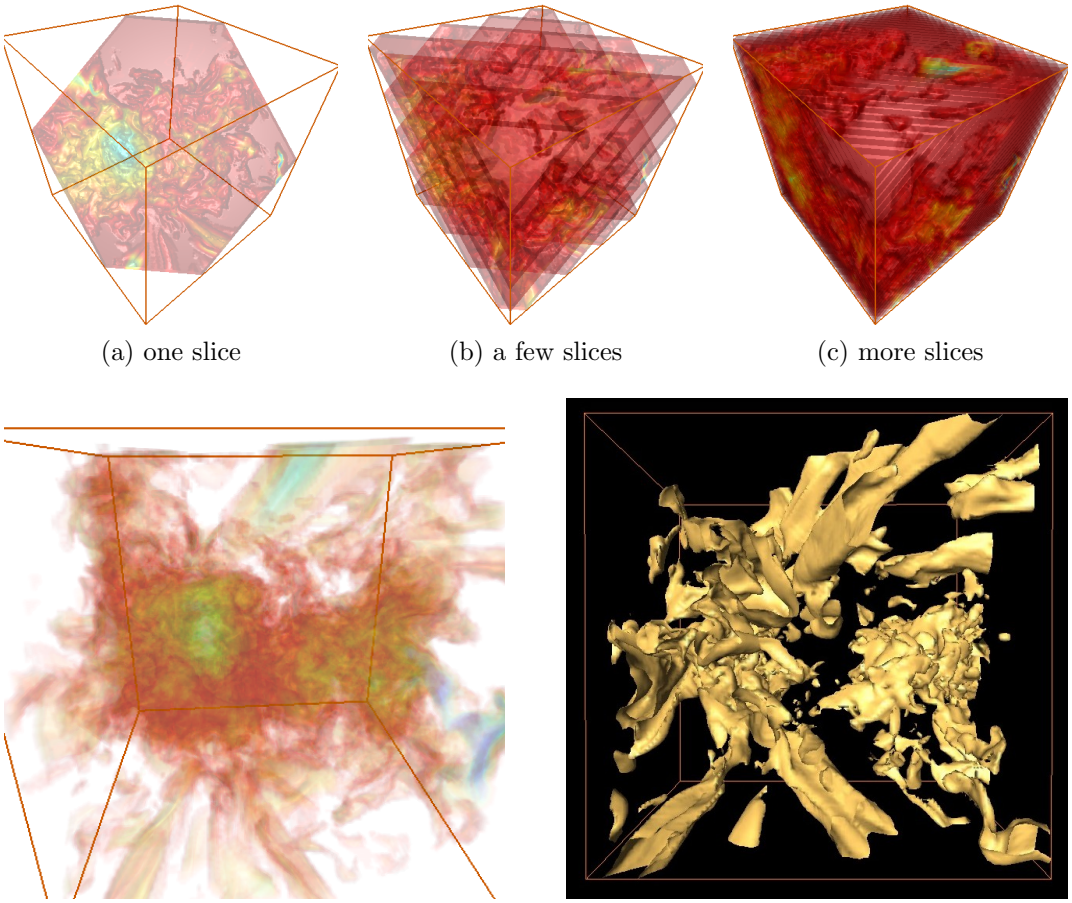
Finally the pixel values of the slices are summed up to create the final shaded volumetric view. For this we define a factor α with $0 < \alpha < 1$, which ensures that the slices closer to camera give a higher contribution than those further away. This is equivalent to the light emitted by the slice travelling through an absorbent medium with an absorption coefficient $\frac{\ln(1-\alpha)}{\Delta\lambda}$. In practice values around $\alpha = 0.04$ for $n_{\mathrm{sl}} = 512$ have shown very good results with good insight into the centre of a data cube. The final image P_{final} is summed up according to

$$P_{\mathrm{final}}(\xi, \upsilon) = \alpha(1-\alpha)^{n_{\mathrm{sl}}} \sum_{i=1}^{n_{\mathrm{sl}}} \frac{P_i(\xi, \upsilon)}{(1-\alpha)^i}. \tag{9.12}$$

This process of summing up the slices for the final image is demonstrated in Fig. 9.4. An example for data visualised with the method discussed here is shown in Fig. 9.4d.

9.4 Results and Discussion

The advancing need for numerical simulations in science, engineering, medicine and many other fields leads to ever increasing amounts of data that have to be dealt with. It proves to be a challenge to analyse and visualise what now often are Terabytes of 3D data in reasonable time and quality. Therefore it is important to both develop new methods of visualisation and improve on already existing methods. The method described in this paper is suitable for visualising volumetric data in real-time and is primarily limited by the amount of graphics memory and the filling rate of the texture mapping units (TMU), which both are quickly expanding in every new generation of graphics cards. With a moderately current graphics card (e.g. a NVIDIA GeForce GTX 285) we are able to visualise a cube of 1024^3 data points with 512 slices at fullscreen resolution (1680x1050) at interactive frame rates (> 30 fps).



(a) one slice (b) a few slices (c) more slices

(d) final image (e) iso-surface

Figure 9.4: An example of the presented method being applied to a volumetric data set showing the metal distribution in galaxy cluster. The final image is composed of fully texturised and shaded slices which are stacked over each other. The diffuse and specular light coming from the slices experiences an absorbent medium on its way towards the camera. Therefore slices close to the camera contribute more to the final image than those far away. The number of slices increases from (a) to (d). In (d) values below the threshold $D_{\min}$ were set to be fully transparent (i.e. $C_\alpha(0) = 0$), which allows a deeper look into the volume. Even when viewing through the surrounding material (red) the inner structure (green) in the centre retains a perception of depth and surface structure. For comparison a polygon based iso-surface is shown in (e).

The advantages of the proposed shading method over unshaded volume renderings can be clearly seen in Fig. 9.2 and 9.4e. Due to the high performance of this

technique, it can also be used in conjunction with a stereoscopic screen or projection to further improve the three-dimensional impression. In fact our visualisation tool *X_View*[4] is already capable of this combined approach.

Acknowledgements

This work was supported by the Austrian Ministry of Science BMWF as part of the UniInfrastrukturprogramm of the Forschungsplattform Scientific Computing at LFU Innsbruck. We thank the anonymous referees for their valuable comments.

[4]To obtain a *X_View* binary please write an email to tobias.riser@uibk.ac.at.

Bibliography

[Angel, 1990] Angel, E. (1990). *Computer Graphics*. Addison-Wesley.

[Cook & Torrance, 1982] Cook, R. L. & Torrance, K. E. (1982). A reflectance model for computer graphics. *ACM Trans. Graph.*, 1, 7–24. URL: http://doi.acm.org/10.1145/357290.357293.

[Ebert & Rheingans, 2000] Ebert, D. & Rheingans, P. (2000). Volume illustration: non-photorealistic rendering of volume models. In *Proceedings of the conference on Visualization '00*, VIS '00 (pp. 195–202). Los Alamitos, CA, USA: IEEE Computer Society Press. URL: http://portal.acm.org/citation.cfm?id=375213.375241.

[Foley et al., 1994] Foley, J. D., Phillips, R. L., Hughes, J. F., van Dam, A., & Feiner, S. K. (1994). *Introduction to Computer Graphics*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[Glassner, 1989] Glassner, A. S. (1989). *3D Computer Graphics*. Design Books, second edition.

[He et al., 1991] He, X. D., Torrance, K. E., Sillion, F. X., & Greenberg, D. P. (1991). A comprehensive physical model for light reflection. *SIGGRAPH Comput. Graph.*, 25, 175–186. URL: http://doi.acm.org/10.1145/127719.122738.

[Kapferer et al., 2007] Kapferer, W., Kronberger, T., Weratschnig, J., Schindler, S., Domainko, W., van Kampen, E., Kimeswenger, S., Mair, M., & Ruffert, M. (2007). Metal enrichment of the intra-cluster medium over a Hubble time for merging and relaxed galaxy clusters. *A&A*, 466, 813–821.

[Kapferer & Riser, 2008] Kapferer, W. & Riser, T. (2008). Visualization needs and techniques for astrophysical simulations. *New Journal of Physics*, 10(12), 125008–+.

[Kniss et al., 2003] Kniss, J., Premoze, S., Ikits, M., Lefohn, A., Hansen, C., & Praun, E. (2003). Gaussian transfer functions for multi-field volume visualization. *Visualization Conference, IEEE*, 0, 65.

[Phong, 1975] Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, 18(6), 311–317.

[Westermann & Sevenich, 2001] Westermann, R. & Sevenich, B. (2001). Accelerated volume ray-casting using texture mapping. In *Proceedings of the conference on Visualization '01*, VIS '01 (pp. 271–278). Washington, DC, USA: IEEE

Computer Society. URL: `http://portal.acm.org/citation.cfm?id=601671.601713`.

Article 10

Visualization of Data from Integral Field Spectroscopy and the P3d Tool

Martin M. Roth, Christer Sandin
innoFSPEC Center for Innovation Competence
at Astrophysikalisches Institut Potsdam
An der Sternwarte 16, D-14482 Potsdam, Germany
mmroth@aip.de

Integral Field Spectroscopy is a powerful observing technique for Astronomy that is becoming available at most ground-based observatories as well as in space. The complex data obtained with this technique require new approaches for visualization. Typical requirements and the p3d tool, as an example, are discussed.

10.1 Integral Field Spectroscopy

Integral Field Spectroscopy (IFS) [1]FS is – somewhat confusingly – also called "3D" or "tri-dimensional" spectroscopy, "two-dimensional" or "area" spectroscopy, also "hyperspectral imaging" is a powerful observing technique that has been introduced and refined over the past two decades. It is now at the verge of becoming a standard tool, which is available at most modern telescopes [Roth(2010)]. For practical reasons, some users have, furthermore, adopted the intuitively descriptive terminology "3D" as a reference to the datacube, which is thought to be the product of an observation (cf. Fig. 10.1).

[1]I

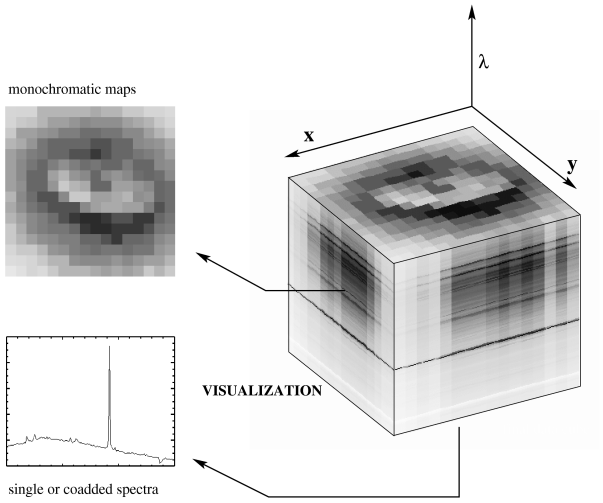


Figure 10.1: Three-dimensional dataset as the result from IFS (there are two spatial coordinates and one wavelength coordinate). The datacube can be visualized as a stack of quasi-monochromatic images, or, alternatively, as an assembly of $n \times m$ spectra.

IFS is an astronomical observing method based on the creation of a single exposure spectra of (typically many) spatial elements ("spaxels") simultaneously over a two-dimensional field-of-view (FoV) on the sky. Owing to this sampling method, each spaxel can be associated with its individual spectrum. Once all of the spectra have been extracted from the detector frame, in the data-reduction process, it is possible to reconstruct maps at arbitrary wavelengths. For instruments with an orthonormal spatial sampling geometry, the spectra can be arranged on the computer to form a three-dimensional array, which is most commonly called a "*datacube*". Datacubes are also well-known as the natural data product in radio astronomy. However, there are many integral field spectrographs which do not sample the sky in an orthonormal system. In this case the term datacube is misleading. Also, atmospheric effects, in particular in the optical wavelength regime, make the term inappropriate in the most general case.

Instruments that create three-dimensional datasets in the above mentioned sense, however not simultaneously but rather in some process of sequential data acquisition (scanning) – e.g. tunable filter (Fabry-Perot) instruments, scanning long-slits, etc. – are not strictly 3D spectrographs according to this definition.

Image Dissection, Spatial Sampling, Spectra

Integral field spectrographs have been built based on different methods of dissecting the FoV into spaxels, e.g. optical fiber bundles, lens arrays, optical fibers

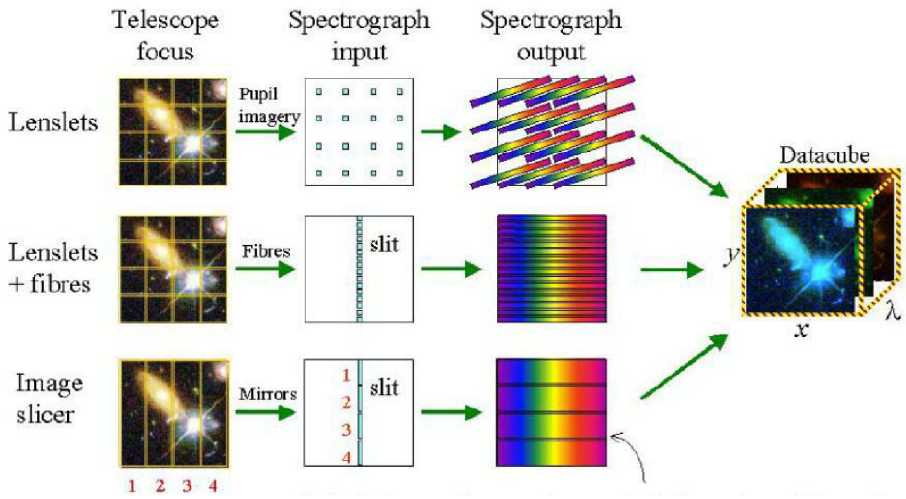


Figure 10.2: The three major principles of operation of present-day IFUs (source: J.Allington-Smith).

coupled to lens arrays, or slicers (Fig. 10.2).

The term *spaxel* was introduced in order to distinguish spatial elements in the image plane of the *telescope* from *pixels*, which are the spatial elements in the image plane of the *detector* [Kissler-Patig et al.(2004)]. The optical elements that accomplish the sampling of the sky are often called "integral field units" (IFUs), and IFS is also sometimes called "IFU spectroscopy". Spaxels can have different shapes and sizes, depending on instrumental details and the type of IFU (Fig. 10.3).

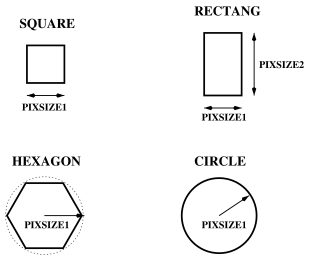


Figure 10.3: Four different types of spaxel geometries: square, rectangular, hexagonal, and bare fiber (circular).

Contrary to the persuasive implication of the datacube picture, IFUs do not necessarily sample the sky on a regular grid: e.g. fiber bundles, where, due to the manufacturing process, individual fibers cannot be arranged to arbitrary precision. Even if the manufacture of an IFU allows to create a perfectly regular sampling pattern, e.g. in the case of a hexagonal lens array, the sampling is not necessarily orthonormal. Moreover, real optical systems create aberrations and, sometimes,

non-negligible field distortions, in which case the spectra extracted from the detector do not sample an orthogonal FoV on the sky. Furthermore, the sampling method may be contiguous (e.g. lens array) fith a fill factor very close to unity, or non-contiguous (e.g. fiber bundle) with a fill factor of significantly less than 100%. In all of these cases, it is possible to reconstruct maps at a given wavelength through some process of interpolation and, repeating this procedure over all wavelengths, to convert the result into a datacube. Note, however, that interpolation often produces artifacts and generally involves loss of information.

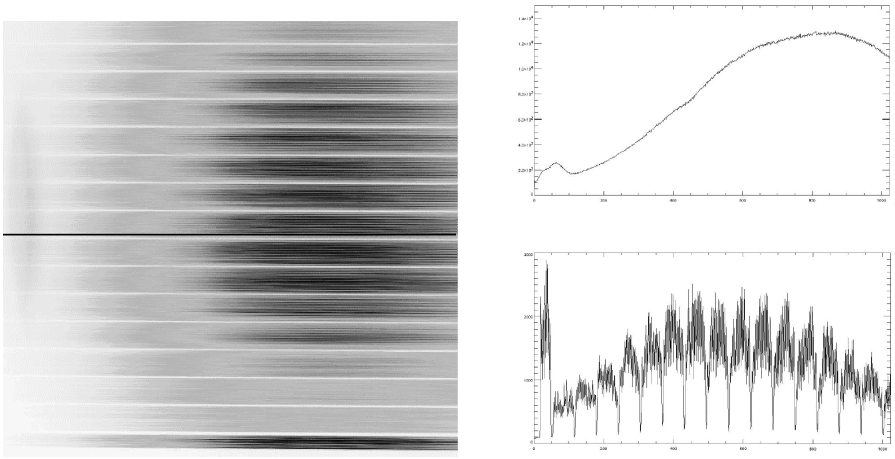


Figure 10.4: Extraction of IFU spectra from a raw CCD frame.

Data Formats

The generic data product from IFS is a set of spectra, which are associated with a corresponding set of spaxel positions. The spaxel coordinate system may or may not be ortho-normal, but in the most general case it is not. It is only through the process of interpolation in the spatial coordinate system that arbitrary IFU geometries are converted to ortho-normal, i.e. a datacube compatible form. Interpolation, however, inevitably incurs loss of information. Therefore the Euro3D consortium has introduced a special data format for transportation of reduced 3D data which is different from the seemingly simple application of the standard FITS NAXIS=3 format, which is suitable e.g. for radio astronomy [Wells et al.(1981)]. The Euro3D data format [Kissler-Patig et al.(2004)] avoids this latter step of interpolation and assumes only that the basic steps of data reduction have been applied to remove the instrumental signature, but else presenting the data as a set of spectra with corresponding positions on the sky. This approach leaves spatial interpolation and the creation of maps the process of data visualization and analysis, i.e. under con-

trol of the user. Figure 10.5 illustrates the spaxel-oriented approach of the Euro3D FITS data format.

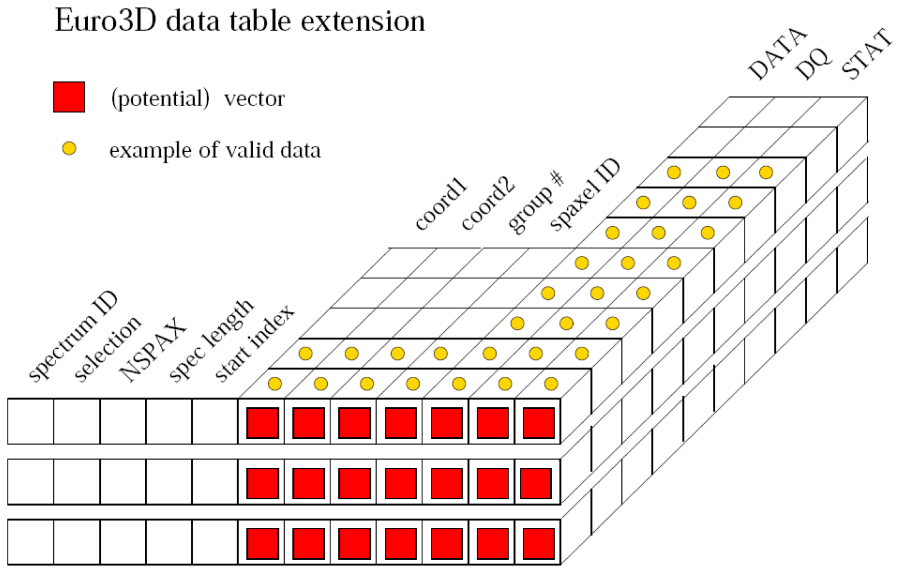


Figure 10.5: The Euro3D data format [Kissler-Patig et al.(2004)].

10.2 IFS Visualization

The visualization of IFS data is confronted with two fundamental requirements: the *inspection* of data for the purpose of monitoring data quality and correcting defects, and the *analysis* of data, i.e. the derivation of physically meaningful quantities. Ideally, a visualization tool should support both. While the former issue requires to preferentially look into *basic elements* of a dataset – for example to identify detector faults that mimic a signal, to check the correctness of the various calibration steps (bias subtraction, flat-field and wavelength calibration, extraction of spectra) and so forth – the latter addresses several possible projections of the data set. For example, the user is often interested in obtaining a *map* at one or several wavelengths over the FoV, corresponding, for example, to emission lines of an extended gaseous object – in order to create line ratio maps, from which one can derive quantities like electron temperature, density, dust extinction etc. On the other hand, for a subset of spaxels that cover a peculiar object, one might wish to co-add the flux within a user-defined aperture, and plot the resulting *spectrum*.

The p3d software [Sandin et al.(2010)] is a versatile data reduction package for optical fiber based 3D instruments, which contains a visualization tool that supports a variety of these needs. Its capabilities are illustrated in Figs. 10.6–10.9.

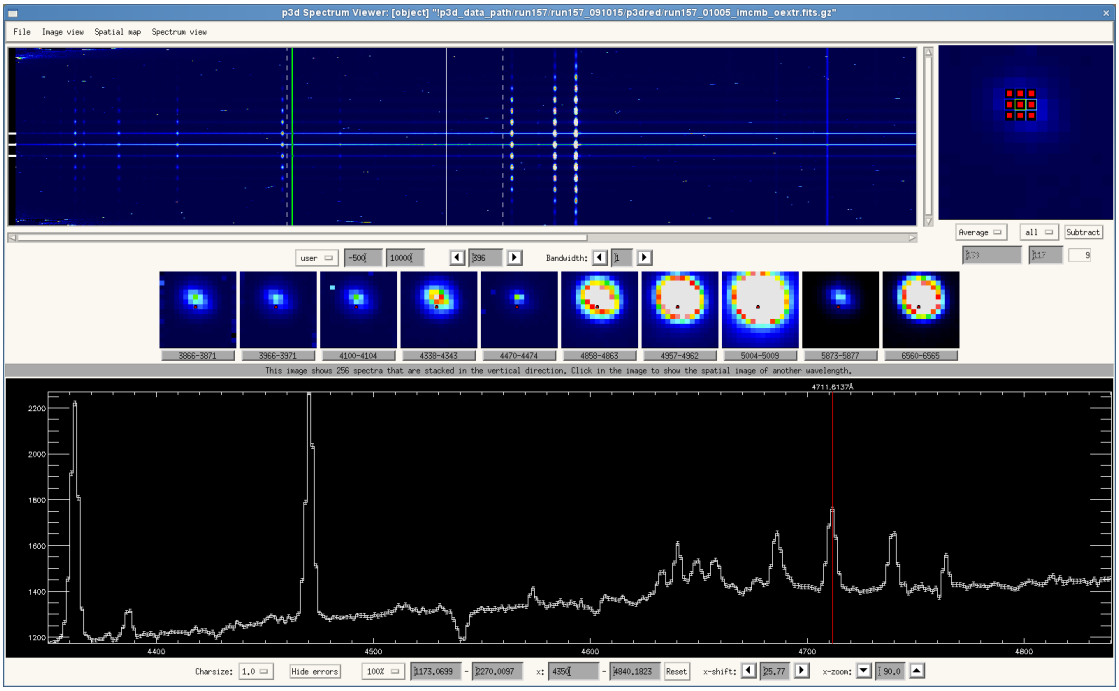


Figure 10.6: A screenshot of the p3d spectrum viewer for a planetary nebula. The different regions of the tool show the spectrum image (in the top-left region, cf. Fig. 10.7), a spatial map of a selected wavelength (in the top-right region, cf. Fig. 10.8), a set of ten stored spatial maps for ten wavelengths and a status line (middle region), and an average plot of the nine spectra that are selected in the spatial map (in the bottom region, cf. Fig. 10.9).

p3d is a free distribution that is licensed under GPLv3. It is available from the project website at `http://p3d.sourceforge.net/`. Although p3d is coded using the Interactive Data Language (IDL) it can be used with full functionality without an IDL license.

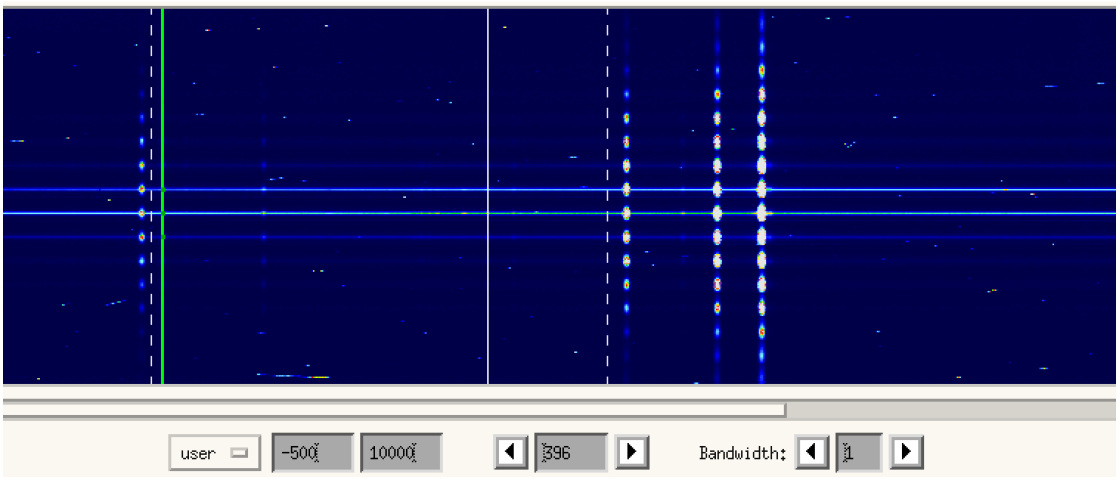


Figure 10.7: Each line in this image represents a spectrum. The brighter features, which are ordered vertically, represent emission lines of Hγ, Hβ, and two forbidden lines of O^{2+} ([OIII] $\lambda\lambda$4959, 5007). The brighter horizontal lines contain the central star continuum, and the randomly placed features are residuals of cosmic rays. The controls at the bottom allow to set the color cut levels, select the wavelength bin that is used to show the spatial map (the green vertical bar), and to define the width of the green bar.

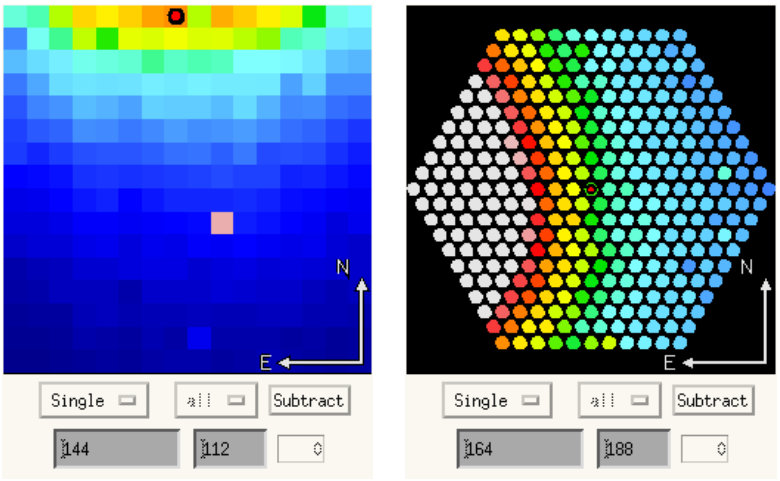


Figure 10.8: The spatial map presents an intensity image as it is seen on the sky. The left-hand (right-hand) side image shows one) selected spatial element, among all 256 (331) square-shaped (circular) spatial elements of the PMAS/LARR (PMAS/PPAK) IFU. The orientation of the IFU is also indicated, north is up and east to the left. The controls at the bottom show the spectrum id, among other things.

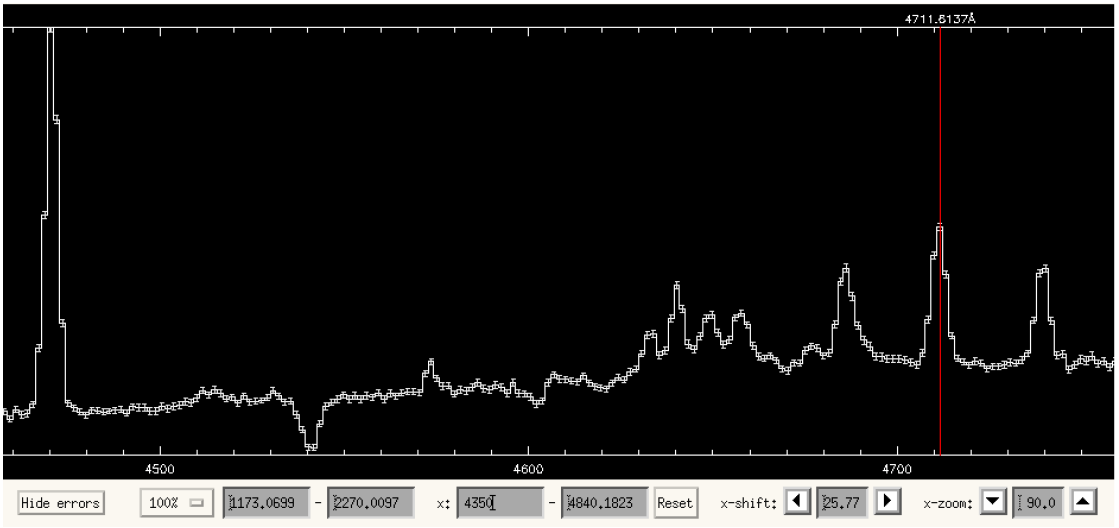


Figure 10.9: This image shows a section of the full wavelength range of the data for the selected spectrum, or, alternatively, of a set of averaged or summed spectra. Errors are shown with bars. The emission line at the cursor location, which is indicated with a red vertical line here (and simultaneously with a white vertical line in the spectrum image, Fig. 10.7) comes from a forbidden line of Ar^{3+}. The controls at the bottom allow a quick change of the properties of the shown spectrum.

Bibliography

[Kissler-Patig et al.(2004)] Kissler-Patig, M., Copin, Y., Ferruit, P., Pecontal-Rousset, A., Roth, M.M., 2004, AN 325, 159

[Roth(2010)] Roth, M.M. 2010, in: *3D Spectroscopy for Astronomy, Lectures of the XVII. IAC Winterschool of Astrophysics*, ed. E. Mediavilla, S. Arribas, M.M. Roth, J. Cepa-Nogue, F. Sanchez, Cambridge University Press, p.1

[Sandin et al.(2010)] Sandin, C., Becker, T., Roth, M. M., Gerssen, J., Monreal-Ibero, A., Böhm, P., Weilbacher, P. 2010, A&A 515, 35

[Wells et al.(1981)] Wells, D.C., Greisen, E.W., Harten, R.H. 1981, A&AS 44, 363

Appendix A

List of Reviewers

- Werner Benger, Center for Computation & Technology at Louisiana State University (CCT/LSU), Baton Rouge, Louisiana, USA
- Andreas Gerndt, German Aerospace Center, Braunschweig, Germany
- Simon Su, Visualization Engineer, High Performance Technologies, Inc. / NOAA-GFDL, Princeton, NJ, USA
- Wolfram Schoor, EADS Deutschland GmbH, Manching, Germany
- Michael Koppitz, Max-Planck Institute for Gravitational Physics, Potsdam, Germany
- Wolfgang Kapferer, Institute for Astro- and Particle Physics, Innsbruck, Austria
- Hans-Peter Bischof, Rochester Institute of Technology, USA
- Massimo Di Pierro, University of DePaul, USA
- Marcel Ritter, Department of Hydraulic Engineering, University of Innsbruck, Austria
- Bidur Bohara, Louisiana State University (CCT/LSU), Baton Rouge, Louisiana, USA
- Edwin Matthews, Louisiana State University (CCT/LSU), Baton Rouge, Louisiana, USA
- Markus Haider, Institute for Astro- and Particle Physics, Innsbruck, Austria

Appendix B

Previous High-End Visualization Workshops

B.1 The 4^{th} High-End Visualization Workshop

The 4^{th} High-End Visualization Workshop was held June 18-22, 2007 in Obergurgl, Austria. Louisiana State University in Baton Rouge, Louisiana, U.S.A. It was featuring the topic of *Visualization of Non-Trivial Data Structures* (ISBN 978-3-86541-216-4, `http://www.lob.de/isbn/9783865412164`). The proceedings included the following contributions:

1. *Modeling of Non-Trivial Data Structures with a Generic Scientific Simulation Environment*, Rene Heinzl, Phillip Schwaha, Carlos Giani, Siegfried Selberherr

2. *Visualization Tools for Adaptive Mesh Refinement Data*, Gunther H. Weber, Vincent E. Beckner, Hank Childs, Terry J. Ligocki, Mark C. Miller, Brian Van Straalen, E. Wes Bethel

3. *The Concepts of VISH*, Werner Benger, Georg Ritter, Rene Heinzl

4. *HD Collaborative Framework for Distribute Distance Learning*, Ludek Matyska, Petr Holub, Eva Hladka

5. *Is Visualization Only Data Representation?*, Wolfgang Kapferer

6. *Direct Surface Extraction from Smoothed Particle Hydrodynamics Simulation Data*, Paul Rosenzweig, Stephan Rosswog, Lars Linsen

7. *Visualization of the Gödel Spacetime*, Frank Grave, Thomas Müller, Günter Wunner, Thomas Ertl, Michael Buser, Wolfgang Schleich

8. *GPU Friendly Rendering of Large LIDAR Terrains*, Shalini Venkataraman, Werner Benger, Amanda Long

9. *A Data Transfer Benchmark*, Andrei Hutanu

10. *Views on Fusion*, Alexander Kendl

11. *A Visualization Toolkit for Lattice Quantum Chromodynamics*, Massimo Di Pierro

12. *A Fast CDESSOR-Based Image Retrieval System*, Yung-Kuan Chan, Yi-Tung Liu, Tso-Yu Lioa, Meng-Husian Tsai

13. *Nucleus and Cytoplasm Contour Detector of Cervical Smear Image*, Meng-Husian Tsai, Yung-Kuan Chan, Zhe-Zheng Lin, Yun-Ju Chen, Chien-Shueh Chen, Shys-Fan Yang-Mao, Po-Chi Huang

14. *Visualization of Polynomials Used in Series Expansion*, Phillip Schwaha, Carlos Giani, Rene Heinzl, Siegried Selberherr

15. *Visualization in Geosciences with Paraview and Geowall*, Christoph Moder, Hans-Peter Bunge, Heiner Igel, Bernard Schubert

16. *Some Development Directions and Examples of Comprehensive Scientific Visualization of Results of Mathematical Simulation*, Mogilenskikh D.V., Mogilenskikh E.A., Melnikova S.N., Pavlov I.V., Petunin S.A.

B.2 The 5^{th} High-End Visualization Workshop

The 5^{th} High-End Visualization Workshop was held March 18-21, 2009 at Louisiana State University in Baton Rouge, Louisiana, U.S.A. It was featuring the topic of *Remote and Collaborative Visualization* (ISBN 978-3-86541-330-7, `http://www.lob.de/isbn/9783865413307`). The proceedings included the following contributions:

1. *Envisioning a Standard Image Storage Framework*, Matthew Dougherty

2. *Interactive Large-Scale Volume Rendering* , R. Parys & G. Knittel

3. *Beyond the Visualization Pipeline: The Visualization Cascade*, Werner Benger, Georg Ritter, Marcel Ritter & Wolfram Schoor

4. *Visualizing Quarks*, Massimo Di Pierro

5. *Post-Processing Pipeline Optimization for Interactive Exploration of Multi-Block Turbine Propulsion Simulation Datasets*, Andreas Gerndt, Rolf Hempel, Edmund Kügeler & Torsten Kuhlen

6. *Towards and Interactive and Distributed Visualization System for Exploring Large Data Sets*, Andrei Hutanu, Jinghua Ge, Cornelius Toole, Jr. & Gabrielle Allen

7. *Remote Rendering Strategies for Large Biological Datasets*, Wolfram Schoor, Marc Hofmann, Simon Adler, Werner Benger, Bernhard Preim & Rüdiger Mecke

8. *Strategies for Efficient Transparency Determination Based on Depth Peeling*, Lars Uebernickel, Wolfram Schoor & Bernhard Preim