Parallel rendering within the integrated simulation and visualization framework *gridlib*

Peter Kipfer   Günther Greiner

University of Erlangen-Nürnberg, Computer Graphics Group
Am Weichselgarten 9, 91058 Erlangen, Germany
eMail: `kipfer@cs.fau.de`

Abstract

Over the last years, computational fluid dynamics (CFD) research has developed several advanced numeric methods for simulating fluid transport. The models being used have grown considerably in size and so have the computed results. Computer graphics research has developed efficient methods for visualization and rendering, to create images of the computed result that contain significant information. The whole process of using CFD methods in engineering however involves many iterations through the model-simulation-visualization cycle. When using reasonably detailed models, the whole cycle suffers from delays produced by the necessary data conversion and data transport. We have developed a solution to this problem by designing an object-oriented framework for integrating simulation and visualization. Computation routines are free to use the provided grid management interface or can be integrated on a binary level by specifying the expected memory layout to the framework. Both simulation and visualization algorithms can be run on the parallel computer. The rendering subsystem therefore has access to the full grid resolution to produce images of high visual quality.

Keywords: Parallel Rendering, Visualization, CFD Simulation

1 Introduction

The CFD community has developed two major groups of fluid simulation strategies: finite element solvers for the full Navier-Stokes equation that can operate on unstructured grids [20, 17, 18] and Lattice-Boltzmann methods for fast simulation on regular Cartesian grids [19, 13]. While the latter are relatively new methods working on highly specialized structures, there is a rich choice of methods available for the Navier-Stokes equation working on geometry. The system our renderer is integrated in, is especially intended for geometry based solvers working on unstructured adaptive hybrid grids.

For realistic and practically relevant simulation results to nowadays problems, the model size has grown continuously resulting in dataset sizes that easily surpass the memory available even on larger workstations. Therefore, simulations are normally run on a highly specialized supercomputer which provide high numerical performance and large memory resources. Unfortunately they lack most often a graphical user interface and/or hardware accelerated graphics boards for rendering. Therefore, the simulation results must be transferred back to the local workstation for visualization. Because of the size of the computed result dataset, reduction methods must be applied to be able to handle the result with the limited resources of the workstation. This is time consuming and can introduce significant errors [4, 16].

1.1 Related Work

Our approach is to integrate both the simulation and visualization process on the simulation machine. Apart from data transfer considerations, this gives the visualization subsystem access to the full detail of the simulation grid: The *gridlib* is a object-oriented framework library for building integrated simulation-visualization systems for unstructured adaptive hybrid grids. There is a number of already existing remote visualization systems [5, 15, 14, 4, 11].

What sets the *gridlib* apart from most other approaches is the basic concept of efficient automatic data storage while maintaining direct array access like referencing a `char[]`. Primitive objects are triangles, quadrilaterals, tetrahedrons, hexahedrons, prisms, pyramids, octahedrons, vertices and edges. They really exist as a class in the standard object-oriented sense and can be constructed and destructed with `new` and `delete`. The developer never has to worry about the actual implementation. Furthermore, there is a set of robust algorithmic operators for algorithmic abstraction and integration of high performance procedural programming language concepts and for binary-level integration of commercial CFD code.

The following section provides a short overview of the framework by presenting the two main abstraction concepts. Section 3.1 explains the implementation of the visualization and rendering subsystem. After that, we present some results and conclude in section 5.

2 System overview

The *gridlib* project uses several advanced object-oriented paradigms in order to introduce three major abstraction layers as depicted in figure 1: A layer for grid topology and grid handling algorithms, one layer for abstraction of element type and one layer for abstraction of storage layout. In the following, we review the two most important concepts of storage and algorithmic abstraction that are used to build the three layers [8, 9].

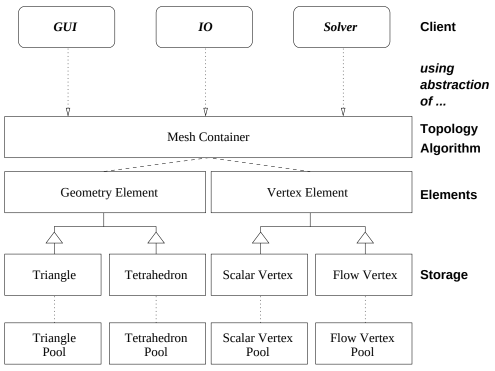

Figure 1: Overview of the *gridlib* system design.

In order to provide abstraction with respect to the memory layout of storage space, the data of each primitive object type is taken care of by a memory-pool. Each pool implements a specific storage layout and packages consecutive items in a linear addressable space. In general, each primitive object type is associated with a specific memory-pool implementation. Note that a memory-pool has no notion of what kind of data is stored in it. The programmer therefore can choose any granularity and ordering of data-packing (sequence of structs vs. struct of sequences).

The very concrete nature of each memory-pool implementation is extended by a design pattern called *external polymorphism* [3, 6] to make use of object-orientation. It builds a virtual inheritance tree externally to the implementations of the participating objects with a pure virtual interface on top of the tree. Method calls are routed by sufficiently global adaptor routines to the actual implementation. Because of the common functionality of the memory-pools, it's very easy to construct these adapters using templates.

Now we are able to view all the diverse memory-pool implementations as related by inheritance, although they aren't, which makes writing higher level object-oriented routines much easier. At the same time we are able to give pointers to the data to procedural third-party code. The memory layout prescribed by the commercial code is maintained by the memory-pool and abstracted for the object-oriented grid handling of the *gridlib*.

The intermediate element abstraction layer in figure 1 is a direct result of the storage abstraction by formulating a common (pure virtual) interface as a parent interface of the abstracted memory layout. The uppermost level provides abstraction of algorithms through a mesh container. It is responsible for encoding topology and geometry of a mesh by exclusively using the abstract element interfaces below. Therefore it's algorithms work on any given storage layout. It acts as a pure container, which means that it's algorithms may not make any assumption about the implementation of the elements or the grid topology.

The second important concept deals with the efficient formulation of algorithms on the mesh level: The mesh container provides algorithmic abstraction. Practically every algorithm dealing with a grid performs `forall` iterations on its elements, vertices or edges. We therefore provide a concept that supports this particular algorithmic pattern. One can write small code entities, called *functors*, that are to be applied to all elements, vertices or edges of the mesh. The mesh container has methods that take any such functor definition, construct the functor and apply it to every contained element, vertex or edge. Because a functor uses the abstract element interfaces below, one can use the full possibilities of overloading, inheritance and runtime type information when writing one.

A functor is implemented by overloading the default operator `operator()` of a lightweight class. The mesh container provides iterators for executing the functor on every grid element. Note that this concept is very useful for parallelization because of the clear separation and encapsulation of the working domain of the functor: Depending on the point in time when a functor is constructed, its private data members reside in shared or thread-specific memory. The actual algorithm that is formulated as a sequence of functor applications to the mesh is called *skeleton program*. The limited actions a single functor performs, along with the expressive power of the skeleton program helps the readability of the code, minimizes the side-effects the small code block can produce and therefore eases maintenance and code reuse.

Also note that a functor must be "sufficiently" global with respect to the skeleton algorithm. A functor therefore can be private to a specific class, can be inherited and serve a whole subtree or can be absolutely global to provide for example a conditional `delete` call on the abstract top-level interface to remove all marked elements, vertices or edges. Putting some generally useful functors into the global scope provides powerful support for code reuse of skeleton algorithms on the mesh level and for providing library functionality to developers.

3 Visualization and Rendering

The integrated visualization and rendering system directly works on the abstraction layers provided and therefore is completely independed of the memory layout and numerical solver used. It features a geometric primitive renderer for transformations, clipping and lighting, from which multiple rasterizers are derived (see figure 3). There are software implementations for all algorithms and OpenGL wrappers for hardware accelerated rendering on screen or into a memory graphics context. This makes three visualization scenarios possible:

- remote rendering on the parallel computer using the high resolution simulation grid by specifying the rendering parameters as command line arguments
- post-processing of the simulation grid, transferring it back to the local high-end workstation and using hardware accelerated local rendering
- hybrid rendering by manipulating a low resolution model on the local workstation, transferring the parameters and rendering the high resolution grid on the parallel machine

Note that especially the last scenario allows running visualization and simulation in parallel and even makes steering of the simulation possible, if supported by the numerical solver [1]. We have implemented the first two scenarios. In this paper we focus on the remote rendering on the supercomputer.

3.1 Visualization

The visualization subsystem works on unstructured hybrid adaptive grids. It offers methods for generating and displaying boundary surfaces, isosurfaces and slices of the grid. Scalar values can be mapped onto the slices. All methods produce triangulated geometries which are handled by the rendering subsystem. After processing all of the triangles by the geometric primitive renderer (compare figure 3), an image can be drawn into any on-/ off-screen rasterization context.

For displaying scalar volume data, the visualization subsystem features a direct volume visualization algorithm for unstructured grids. The unstructured grid is sliced perpendicular to the viewing direction and the created planes are blended during rendering, similar to the idea of volume rendering using 3D texture mapping. The slicing is performed by defining the slicing plane through the grid, determining the intersected cells and calculating the intersection points with the plane by linear interpolation of the two vertices at each intersected edge. Then, the same interpolation is also performed for the scalar values of the edge's vertices. During rendering, a transfer function is applied to the values. In case of on-screen rendering, the transfer function is adjustable interactively (see figure 2).

Of course, the slicing algorithm can also be slightly modified to apply to arbitrary surfaces within the 3D grid for mapping color-coded data. In

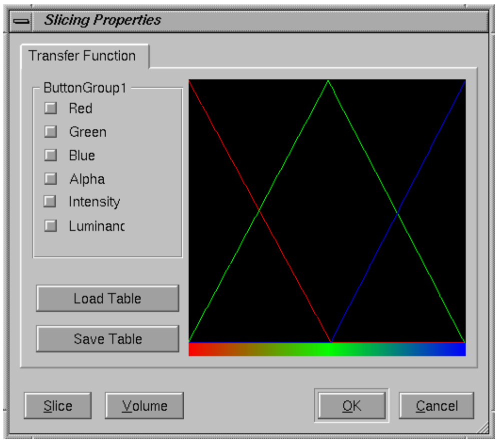

Figure 2: Scalar values can be mapped onto slicing planes by an interactively adjustable transfer function.

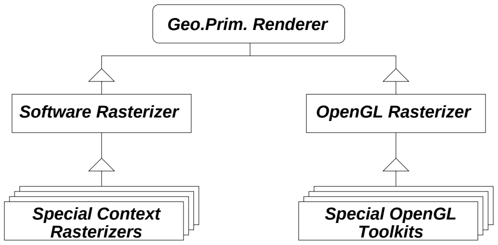

Figure 3: The rendering subsystem derives several software based rasterizers and hardware accelerated implementations from a single well defined interface.

combination with the isosurface extraction method, correlation within the data can be found.

The *gridlib* visualization subsystem features an isosurface extraction algorithm for unstructured grids with progressive surface reconstruction [12]. The algorithm is very fast through using an interval tree [2] to search for the elements intersected by the isosurface. The algorithm uses the grid handling capabilities of the *gridlib* to build a progressive representation of the grid. Note that this can also be used for rendering continuous levels of detail, which is especially useful for keeping the framerate interactive for local hardware accelerated visualization.

3.2 Rendering

The rendering subsystem is split into two major parts: An abstract renderer for geometric primitives and a rasterizer. Currently, there are rasterizer implementations for triangles and lines. There is a pure software rasterizer and a wrapper for OpenGL, both have several derived children for special rendering contexts or hardware accelerated toolkits.

The rendering subsystem is fed by the visualization methods with individual triangles or triangulated meshes. The geometric primitive renderer processes them with standard computer graphics algorithms for culling and clipping in order to reduce the number of triangles to rasterize. It's main task is to serialize the triangles into a specific formatted stream for the rasterizers. This includes computation of vertex attributes like color (probably from a transfer function mapping) and texture coordinates (including loading of texture bitmap). The renderer also features portal culling in order to reduce the overdraw rate of the rasterizers. The geometric primitive renderer is a pure software implementation and can therefore be used for all rasterizers.

For rendering on the parallel supercomputer, the software rasterizer is used. It works similar to OpenGL by using a Z-buffer. Although there is some CPU-cycle penalty for the overdraw, this technique allows to implement the scanline code to treat the Z-component similar to other scanline attributes like color and texture. This gives longer code sequences without branches and therefore better optimization possibilities for the compiler. However the most appealing aspect of the Z-buffer technique is, that the rasterizer can be kept quite simple to implement. It finally provides a framebuffer and the Z-buffer to it's derived children which are left with implementing the transport of the framebuffer content on screen or into an image file.

4 Parallelization and Results

Within the *gridlib* framework, abstraction and encapsulation is performed on several levels. Along with ensuring reentrant implementations, it is the foundation parallelization and distribution is built on. For the actual implementation there is a variety of concepts for running code concurrently, ranging from operating system specific threading to high-level middleware. On supercomputer systems, the

MPI interface has become a quasi-standard. It has a well defined set of routines for exchanging information based upon the message passing paradigm which is a synchronous rendez-vous concept. This makes it possible to use the interface for shared and distributed memory architectures equally. The MPI library itself is often specifically tailored to the supercomputer for optimal performance. The *gridlib* uses MPI for transparent parallelization and distribution. Although the *gridlib* supports Linux on the PC, IRIX on SGI Onyx and Origin machines, we concentrate in this paper on the Hitachi SR8000-F1 supercomputer architecture.

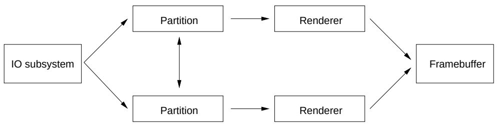

Figure 4: Overview of data flow for rendering.

Figure 4 shows the general data flow to the rendering subsystem: A designated process requests the grid from the IO subsystem and distributes it equally in terms of elements to all participating processes. Then, a refined partition is computed by all processes in parallel. Geometry or custom weighting of the elements can be taken into account (vertices $\mapsto$ grid nodes). Also, the *gridlib* can create the dual grid and compute the partition graph on it (vertices $\mapsto$ cell centroids). After performing the simulation, the partition is reused by the rendering subsystem to draw only the assigned part into a private rendering context. Note that this also distributes the memory requirement. Upon completion, all partial images are combined into the final framebuffer by a synchronized method.

Because the sizes of practically relevant datasets for CFD simulation easily exceed local memory resources, a dataset must be distributed among processing entities (PEs). The interconnect between the PEs has much less bandwidth than local memory and is therefore a bottleneck. Consequently this communication cost dominates the total data exchange time. The *gridlib* framework processes the grid geometrically and topologically in order to build an adjacency graph in parallel. From this graph, the parallel METIS [7] library computes an optimal partition which is evaluated by the *gridlib* for moving the geometry accordingly.

For running CFD solvers, the partitioning criterion clearly has to minimize the number of partition boundary elements in order to minimize the communication cost. The rendering subsystem uses the partition to run the visualization and rasterization in parallel. This does not provide an equal number of triangles to render for each PE, because it depends on the visualization method and parameterization. However, transferring geometry between the PEs is too expensive both in terms of time and memory. Therefore we compute the visualization and rendering on the same partition as the simulation to avoid delays.

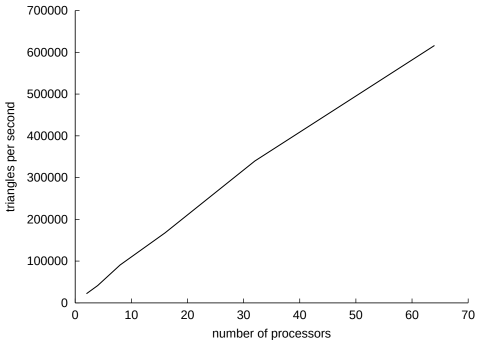

Figure 5: Scaling of the rendering subsystem in terms of number of processors used.

The following results have been obtained on a Hitachi SR8000-F1 supercomputer [10] using a subset of 8 nodes of the machine. Each node is equipped with 9 processors with one being reserved for system activity and has 8 GB of local memory. The distributed resources used in this paper therefore are 64 processors with 64 GB of memory. Although the processors of one node are sharing the memory, the parallelization concept of the *gridlib* performs a one-to-one mapping of processors to processes. This has the advantage of being free to place a process on any processor in any node to ensure data locality and coarse load balancing.

We have evaluated the rendering subsystem implementation for the worst case of rendering all the faces of all elements of the simulation grid. This incurs considerable overdraw (compare figure 10). Many visualization algorithms however output unrelated triangles, which is simulated by this approach. The triangles can be rendered immediately or can be stored in a 2D mesh. Note that a typical visualization as in figure 11 normally has several orders of magnitude less triangles to draw than there are grid elements. Because we are drawing the triangles produced by the visualization directly from the simulation grid, the renderer needs no additional memory apart from the framebuffer and some state information. All timings have been measured in wall-clock time, because this is what users are interested in.

Figure 5 demonstrates the scalability of the rendering subsystem. We get almost linear speedup although the communication volume in the Z-buffer merging stage is growing with the number of processors. Note that we minimize the amount of pixels to transport by projecting the bounding box of the triangles to render into screen space and only selecting the projected rectangle for transport along with its screen space coordinates. The synchronization of the buffer merging stage is ensured by using blocking MPI calls. Although this seems to cause unnecessary delays, it is not a problem in practice, because the overall rendering time in most cases is typically 1 second. Dynamic redistribution of the triangles to rasterize incurs communication overhead that slows down the whole process considerably. The efficiency of our approach is shown in the left experiment of figure 5, where we compare two renderer instances running on the same node with two instances running on two separate nodes. The same was done in the right experiment using four instances on the same node and on four nodes respectively. The performance differences are within the normal measurement jitter using wall-clock time (see figures 6 and 7). The figures clearly show that there is no time penalty for distributing the rasterizers to different nodes: The overhead of the Z-buffer merging stage is not apparent.

When we compare the performance of the rendering subsystem by running 8 rasterizers concurrently, we observe a slight performance loss in the case of running them on a single node (figure 8 left column). This is due to suboptimal MPI support for intra-node operations. In contrast, when we distribute one rasterizer to each one of the 8 nodes, no performance penalty is observed (figure 8 right column). Our profiling on a SGI Onyx (shared memory architecture) machine does not exhibit this behavior and thus clearly identifies the distributed memory

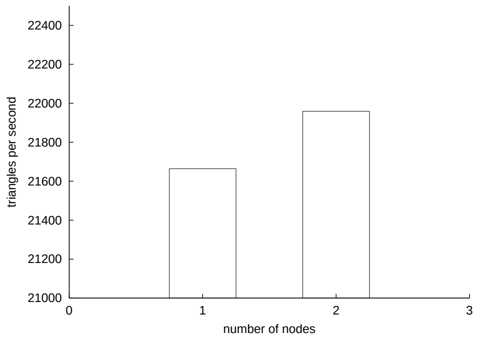

Figure 6: Comparison of assigning 2 renderers to 1 node (left) or 1 renderer to 2 nodes (right).

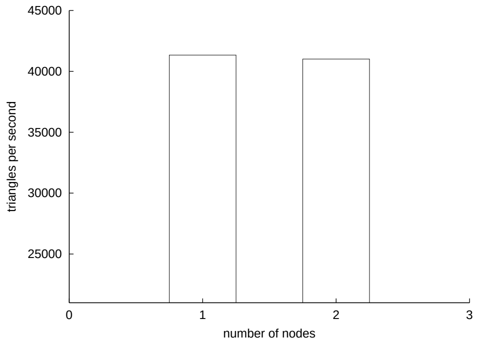

Figure 7: Comparison of assigning 4 renderers to 1 node (left) or 1 renderer to 4 nodes (right).

communication to be the bottleneck of the supercomputer. This also justifies our approach not to compute a separate partition of the grid for rendering: The communication time incurred by it easily exceeds the total time spent for rendering.

As mentioned earlier, we used the worst case of rendering all faces of all elements individually for the above measurements. This can be compared to OpenGL rendering triangle by triangle in immediate mode. Clearly, a lot of optimizations are possible if we allow the renderer to work on a larger 2D triangle mesh, which can be compared to rendering OpenGL triangle-strips in retained mode. This comes at the expense of additional memory requirement for storing the 2D mesh and some acceleration structure. Our rendering subsystem features sev-

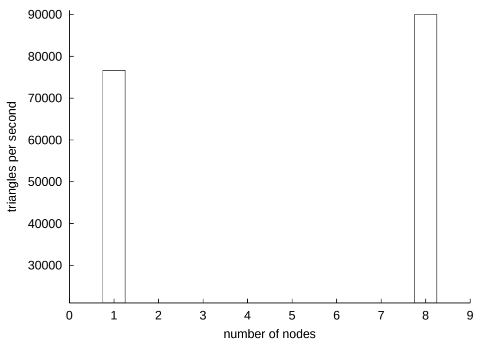

Figure 8: System saturation observed when using all 8 processors of 1 node (left) compared to 1 renderer on 8 nodes (right).

eral standard optimizations for 2D triangle meshes. When using them, the software renderer can produce up to 91.000 smooth-shaded triangles per second on a single CPU.

When performing a typical visualization task like in figure 11, the grid partition automatically breaks down the number of triangles one process has to draw to several thousands. Additionally, we have to draw mostly 2D triangle meshes which enables us to make efficient use of the acceleration techniques of the renderer. So we obtain rendering times of 1-2 seconds on average. Compared to total execution times of the inner simulation-visualization loop of several minutes up to hours, the time spent for rendering is negligible. We can therefore easily afford to render the grid on nearly every intermediate timestep while the simulation is running. The images created can be streamed back to the user interface and help to judge the simulation process to detect divergence, bad boundary or initial conditions in which case the simulation can be aborted to save CPU time.

5 Conclusion

The parallel rendering subsystem of the *gridlib* is consistently integrated into the simulation and visualization framework. It offers one well defined high-level abstract interface and is split into two main components for fragment-based operations and scanline rasterization. The parallelization is based on the message passing concept and makes use of the data distribution computed by the partitioning of the simulation grid. There are several implementations of the rasterizer engine, ranging from hardware accelerated OpenGL-based toolkits to entirely software-based versions. This makes it possible to use it efficiently on both graphic workstations and supercomputers.

We have demonstrated the usage of the parallel rendering subsystem on the Hitachi supercomputer, where it can render the simulation grid at full resolution with little additional memory consumption. The CPU time spent is negligible for most simulation runs compared to the total job time.

5.1 Future work

The parallel rendering subsystem will be extended to allow interactive remote rendering on the supercomputer guided by the user through manipulating a low resolution model on the local workstation. The low resolution model will be generated on the supercomputer on-the-fly from the simulation grid and sent to the workstation using progressive meshing techniques.

For rendering complex spatial relations and gaseous media, we are currently examining the potentials of raytracing. This technique will be used mixed with the fragment-based rasterization and can solve some typical raytracing problems as mentioned in [16].

5.2 Acknowledgments

This work is part of the KONWIHR project “Adhoc3D”, the Competence Network for Technical, Scientific High Performance Computing in Bavaria [10]. The *gridlib* is a joint research project of LGDV (computer graphics group), LSIM (system simulation group) and LSTM (department of fluid mechanics) of the University of Erlangen-Nürnberg. Special thanks go to Manfred Kaltenbacher (department of sensor technology) for providing the datasets.

References

[1] Xavier Cavin, Laurent Alonso, and Jean-Claude Paul. Overlapping multi-processing and graphics hardware acceleration: Performance evaluation. *Proceedings Parallel Visualization and Graphics Symposium (PVG)*, 1999.

[2] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopignio. Speeding up isosurface extraction using inter-

val trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[3] Chris Cleeland and Douglas C. Schmidt. External Polymorphism — An Object Structural Pattern for Transparently Extending C++ Concrete Data Types. *C++ Report Magazine*, September 1998. `http://www.cs.wustl.edu/~schmidt/C++-EP.ps.gz`.

[4] Thomas W. Crockett. An introduction to parallel rendering. *Parallel Computing*, 23(7):819–843, July 1997.

[5] Thomas W. Crockett. Parallel rendering. In *SIGGRAPH '98 "Parallel Graphics and Visualization Technology" course #42 notes*, pages 157–207. July 1998. overview of parallel methods for a variety of rendering algorithms.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addision-Wesley, Reading, MA, first edition, 1995.

[7] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. Technical report, Department of Computer Science and Engineering, University of Minnesota, 1997.

[8] Peter Kipfer. *gridlib*: System design. Technical report, Computer Graphics Group, University of Erlangen-Nürnberg, 2000.

[9] Peter Kipfer. *gridlib*: Numerical methods. Technical report, Computer Graphics Group, University of Erlangen-Nürnberg, 2001.

[10] KONWIHR. Competence Network for Technical, Scientific High Performance Computing in Bavaria. http://konwihr.in.tum.de/index_e.html.

[11] R. Koppler, G. Kurka, and J. Volkert. Exdasy - a user-friendly and extendable data distribution system. *Proceedings EuroPar Conference*, 1997.

[12] U. Labsik, P. Kipfer, S. Meinlschmidt, and G. Greiner. Progressive isosurface extraction from tetrahedral meshes. *Pacific Graphics Conference Proceedings*, 2001.

[13] P. Lallemand and L. Luo. Theory of the lattice-boltzmann method: Dispersion, dissipation, isotropy, galilean invariance and stability. *Physical Review E*, 61, 2000.

[14] A. Law and R. Yagel. The active-ray approach to rendering on distributed memory multiprocessors. Technical report, Ohio State University, 1996. OSU-CISRC-2/96-TR10.

[15] P. Peggy Li, William H. Duquette, and David W. Curkendall. RIVA: A Versatile Parallel Rendering System for Interactive Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):186–201, September 1996.

[16] Kwan-Liu Ma. Parallel visualization of large scale aerodynamic calculations: A case study on T3E. *Proceedings Parallel Visualization and Graphics Symposium (PVG)*, 1999.

[17] S. Riedelbauch, G. Brenner, U. Prinz, and W. Kordulla. Pitfalls and beauties of cfd in hypersonics. *Proceedings, 4th International Symposium on Computational Fluid Dynamics*, II, 1991.

[18] U. Rüde. Stability of implicit extrapolation methods. *Proceedings 8th International Conference on Domain Decomposition Methods*, May 1995.

[19] D. A. Wolf-Gladrow. *Lattice Gas Cellular Automata and Lattice Boltzmann Models*. Springer, 2000.

[20] X. Zhou, Z. Sun, F. Durst, and G. Brenner. Numerical simulation of turbulent jet flow and combustion. *Int. J. Computers & Mathematics with Applications*, 38, 1999.

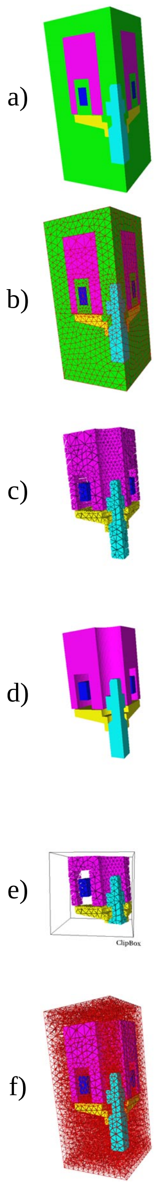

Figure 9: The rendering subsystem has several rendering modes for close examination of detail or interactive rendering of general shape. The images above show a partitioned dataset. Each partition can be turned on/off independently. Each mode or a combination of modes can be applied to each partition separately: a) surface b) surface and edges c) volume elements of adjustable size d) surface of partition e) element-oriented clipping f) volume element edges

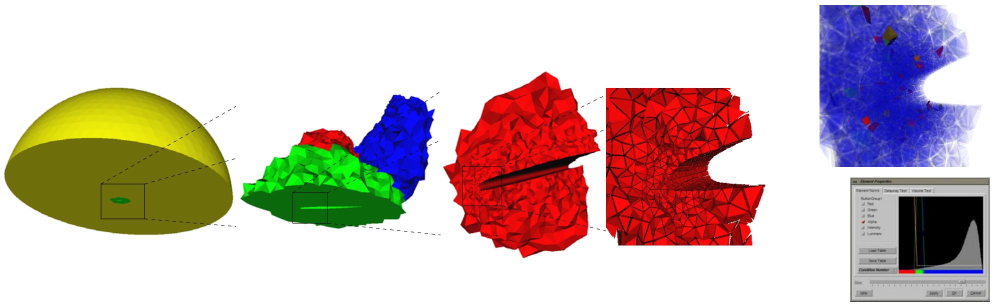

Figure 10: Exploration of a simulation grid: The partitioned and distributed dataset is examined to look for possible meshing errors. Switching off partitions and introducing freely positionable clipping planes allows to keep rendering interactive as we zoom in from left to right until we reach the desired level of detail where we can switch to rendering the elements. Once potentially critical detail is found, element quality can be computed with a variety of methods and visualized on the upper right using a transfer function on the lower right: As the histogram shows, most of the elements are in good shape (according to the criterion) and will be colored transparent blue, but there are some bad ones that will be colored red. Note that the distribution to several processors enables to start from the full model.

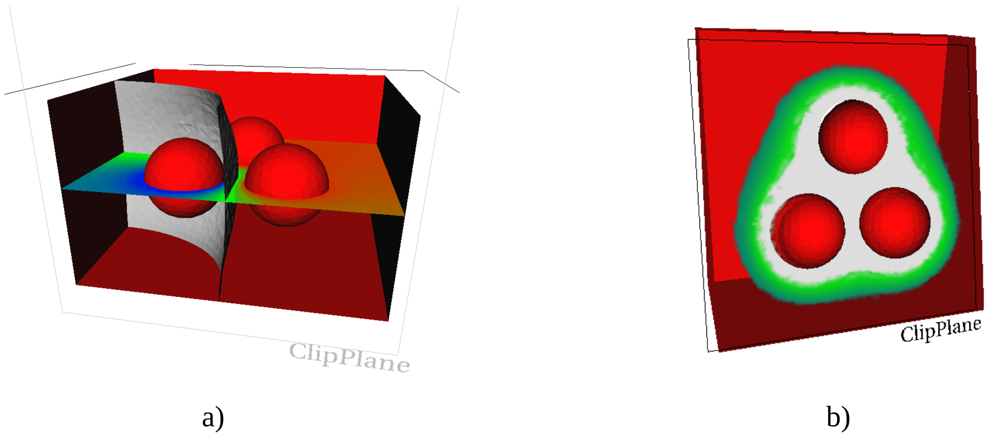

Figure 11: Two examples of possible direct visualization: On the left is an electrostatic simulation showing some special scalar value as an isosurface and a color-mapped slice through the unstructured simulation grid, on the right is the same dataset now with a simulation of heat dissipation. The three spheres are heated and the resulting scalar field is rendered directly from the unstructured grid using volume rendering.