A fast Visualization System for Adaptive Grids*

Gerhard Kurka
e-mail: kurka@gup.uni-linz.ac.at
SFB F013, Project 1301

Abstract

Todays data sets arising from computational mathematics are extremely large which poses increasing demands for performance (in the sense of graphics-performance) on interactive visualization systems. ”Real-time” interaction interfaces have proven useful for the analysis of these data sets, but real-time performance has been difficult to achieve with such large amounts of data. This work describes the outcome of a small visualization project dealing with the visualization of large unstructured data sets.

1 Introduction

At the beginning of our project we faced the problem of integrating a fast visualization system into FE++ [1], a full-fledged finite element package. In order to shorten development time we planed to use GRAPE [3] — a fast visualization package, which provides support for adaptive multilevel grids — as the basis for our visualization system. Unfortunately, there are two serious problems to this approach:

- Software packages using grape for visualization purposes must provide a special program structure, such that GRAPE can take over control of the whole application. FE++ was designed without having in mind GRAPE's application programming interface. Thus its structure doesn't meet the requirements of GRAPE.
- In addition, GRAPE requires a lot of information about the grid and the elements, in order to apply its powerful visualization algorithms. Because FE++ doesn't provide many of these data structures demanded by GRAPE, we would have been enforced to rewrite big parts of FE++ to provide the required data structures in an efficient way.

Due to the notable effort in solving these two problems we decided to develop our own visualization system from scratch. Our goal was to meet following requirements:

- Fast interactive visualization of thousands of elements
- Low memory requirements
- Direct integration into FE++

In the following we briefly describe how these requirements have been fulfilled. In addition, a short overview of the functionality will be given.

*Acknowledgments: This work is supported by the Austrian Science Fund – 'Fonds zur Förderung der wissenschaftlichen Forschung' – within the SFB F013 ”Numerical and Symbolic Scientific Computing”.

2 Accelerated Rendering by means of Multiple Levels of Detail

Maintaining a high interactive frame rate – we mean a minimum of 7 frames per second – places difficult constraints to many layers of a visualization system. A well known approach to achieve interactive visualization, even for very large data sets, are multiresolutional visualization methods [11].

These methods deal with controlling the resolution, or level of detail (LOD), at which objects are rendered. The selection of the object resolution depends to a high degree on the application field. For example, in a virtual reality environment distant objects that appear smaller on the screen are rendered at a lower level of detail than near objects. This strategy can be refined in various ways, e.g. Canon has developed a gaze controlled LOD-method in which objects that appear near the visual focus of the viewer are rendered at a higher resolution than objects rendered in the peripheral regions of the viewer's field of vision.

In other applications the LOD selection depends only on the frame rate. I.e., if the frame rate drops under a predefined limit a subset of objects that the scene consists of will be replaced by coarser ones, in order to reduce the overall complexity of the scene and thereby increasing the frame rate. A common problem in using these techniques is the generation of different resolution forms of a given object. Since instantaneous switching between LOD meshes may lead to perceptible "poping", one often like to construct smooth visual transitions, geomorphs, between meshes at different resolutions. This process can be a very expensive and automatic algorithms are still in the research phase [10, 9].

The hierarchical structure in the multigrid data representation of FE++ allows us to avoid the generation of different resolution stages of the objects to be visualized. Instead, we make direct use of the multigrid data structures which already describe the objects at different resolution stages. This approach reduces the development effort but leads to visual discontinuities appearing when switching between different object resolutions. Althought this is a serious problem in applications like virtual reality environments, it is acceptable in our case, because we are interested in a fast interactive visualization of huge data sets rather than in a perfect illusion of a virtual world.

In essence, our strategy for selecting a LOD looks as following. When the object is drawn first the system selects the coarsest object resolution available. If the rendering time is short enough to ensure at least a minimum frame rate, the system will switch to a finer object resolution. If the frame rate stays within a predefined interval, no further changes to the object resolution will be made. After user interactivity stops, e.g. the user release the mouse button, the system stays idle for about 500ms. Thereafter, it switches automatically to the finest object resolution.

Our experience gained from applying the visualization system to different finite element grids has shown that by using this switching strategy, we don't require tuned visualization algorithms in most cases. Thereby, the visualization system could be kept simple, which results in a drastic reduction of development time to only four month required by single person.

3 Tighly Coupling to FE++

When dealing with three dimensional finite element/finite volume grids composed of hundred thousands of elements it is crucial to keep an eye on the memory requirement. Conventional visualization systems are often using file interfaces to access the data structures to be visualized. I.e., the application data has to be stored in huge files on the disk. From our point of view there are two disadvantages to this approach:

- Creation and processing of large data files will take a lot of time, which leads to notable latencies between providing the solution and presenting it to the user. Especially, when experimenting with different numerical schemes such wait states are often not reasonable to the user.
- Since the solver and the visualization system are both using their own internal representation

of the grid and solution data, a large main memory is necessary to avoid swapping and out-of-core visualization.

In order to solve both problems stated above we have developed the visualization system as part of FE++. I.e., the visualization system is able to access the FE++ grid data base rather than maintaining a redundant copy of it. We have also tried to minimize modifications of FE++ that arise due to the integration of the visualization system. To achieve this goal the visualization system was implemented as a so called thread executing within the context of the FE++ process. Since the visualization thread is able to access all global variables and data structures of FE++ directly, we were able to implement the interface between the two systems by an easy to understand C++-class. In addition, because of the concurrent execution of the visualization thread side on side with FE++ it becomes possible to interact with the visualization system while FE++ is calculating the next solution step with respect to explicit dynamic simulations.

4 User Interface

In general, implementing a user interface is a time consuming job. For this reason various toolkits and application frameworks have been developed to free programmers from recurring and bothering tasks related to user interface programming.

We have chosen the application framework Qt [4], which has proven very useful in many public projects, by the way, it is free for non commercial purposes. Qt is an object oriented application framework entirely written in C++. It supports the development of platform independent user interfaces. In addition, it provides explicit support for 3D graphics programming with OpenGL.

OpenGL [2] is a low level graphics API developed by Silicon Graphics in order to provide a portabel 3D-graphics interface to the graphics hardware. OpenGL is supported by a wide variety of companies, this makes it the ideal graphics API for developing fast and portable visualization systems.

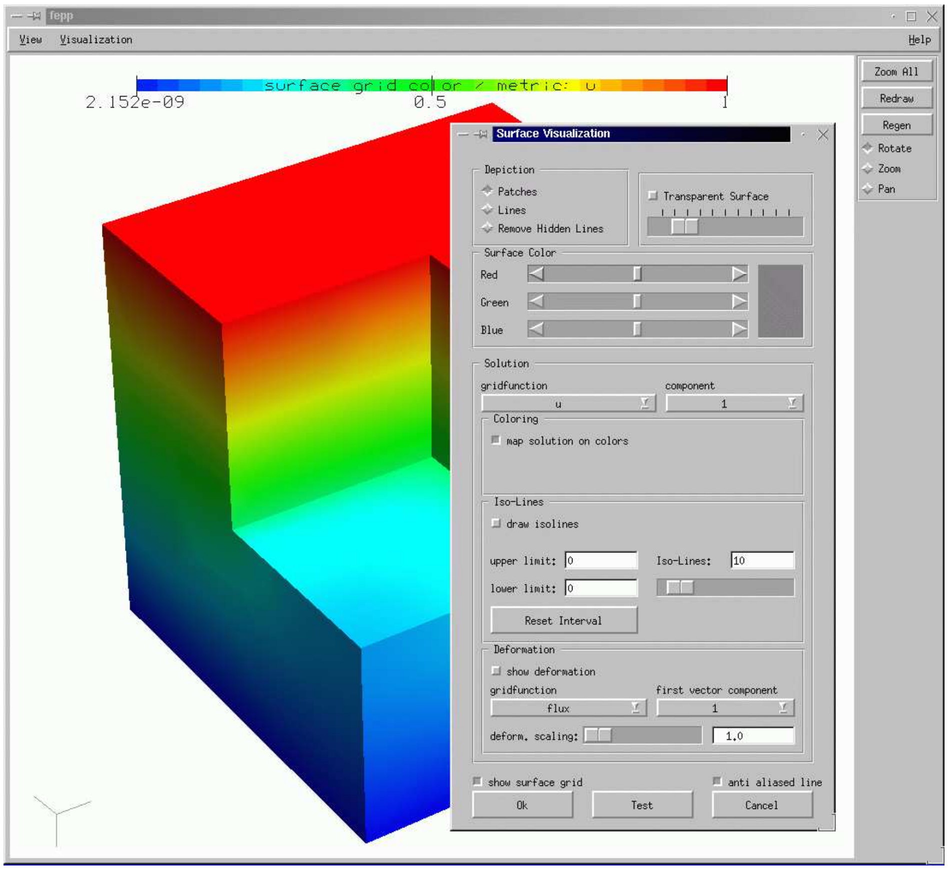

Figure 1: Overview of the user interface

5 Visualization Techniques

This section presents all visualization techniques offered by our system. For further information consider [5][6][7].

5.1 Color Mapping

Color mapping is a scalar visualization technique in which a surface is continuously colored in accordance with a scalar field. It is applied to either polygonal surfaces, or on cutting planes, which are decomposed into triangles. The scalar mapping is implemented by mapping the scalar value on the angle of the HSV color model (Referring Foley Van Damme). The upper and lower colors correspond to the maximum and minimum scalar values. The scalar values at most grid vertices will lie in-between these limits, and the associated color is determined by linear interpolation. After the colors (RGB components) and surface normals (for lighting) are determined for each vertex on the surface, the Gouraud shading model is used to smoothly fill the polygons (see figure 2). Generally, Gouraud shading is used because it is implemented in hardware on most graphics workstations.

5.1.1 Optimizations

Most of the computationally expensive work in this technique, the Gouraud shading, is performed in hardware. A typical dataset consists of 50 000 triangles on the surface. For interactive applications requiring updates at 10 frames/sec, the hardware must be capable of rendering 0.5 million Gouraud shaded polygons per second. The rendering performance has been further optimized by using OpenGL vertex arrays and display lists, which enable the graphics hardware to render nearly at peak performance, e.g. 2 million triangles per second on an Octane/SSE.

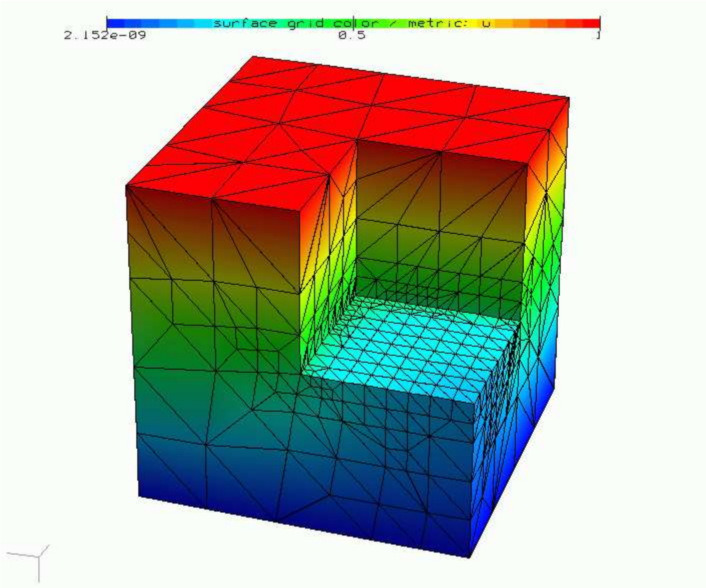

Figure 2: Visualization of scalar values by color mapping

5.2 Contour Lines

Contours are curves where a given scalar variable, f, is constant: f(x,y)=c. As with the color maps, they are usually rendered on a surface grid or on a cutting plane (see figure 3), the former being more efficient. The geometry for a contour line is computed one cell at a time by comparing the scalar values at the vertices to a particular threshold value. A contour line crosses a polygon where the scalar function changes from below to above the threshold value. The intersection points on the edges are determined by linear interpolating the discrete vertex values, and then connected with a line segment.

Our implementation is based on the marching algorithm. The marching technique examines every cell on the surface, in no particular order, and compares them to a case table. The case table contains all the possible topologies for a contour crossing a polygon, assuming that the underlying interpolation function is linear. For a given cell, the appropriate entry is determined by computing an index which encodes the state of the vertex as a binary digit. Once the proper case is selected, the intersection point of the contour line with the cell edge is determined using linear inpolation.

Linear interpolation is widely used because it is simple and efficient. Higher order interpolations produce visually better curves and more accurate results, although they are more expensive and require special attention at boundaries.

5.3 Isosurfaces

An isosurface or contour surface is the 3D equivalent of a contour line – it is a surface that corresponds to a scalar value in a 3D scalar field. Marching algorithms on either hexahedrons or tetrahedrons are widely used.

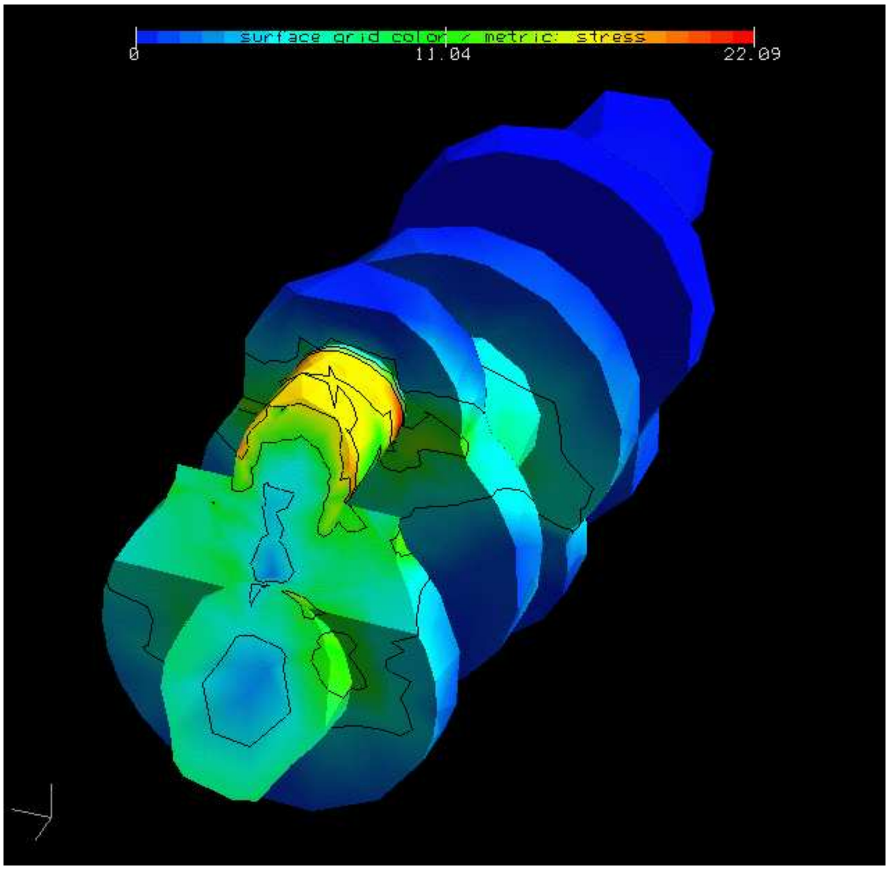

Figure 3: Contour lines combined with color mapping

The isosurface algorithm generate global isosurfaces, i.e., it find every cell which contains a particular threshold and generate the appropriate polygons. Figure 4 shows an example of an isosurface used to visualize the distribution of a certain stress value within a crankshaft.

For CFD applications, isosurfaces can be used to display important flow features such as shock waves.

5.3.1 Pitfalls

Many of the anomalies that occur in 2D contouring, such as scalloping and bumps, also occur on isosurfaces, although these are less evident on shaded and lighted surface. Again, the issue is the mismatch between the interpolation function and the underlying function of the scalar field. Higher-order schemes may resolve these problems, but are not used because of the added computing and rendering costs.

5.4 Cut planes

A cut plane is generated by the same marching algorithm as for the isosurface. To produce a cutting plane a new scalar function has to be computed for each vertex. It is either the x, y or z coordinate for an axis aligned cutting plane, or $ax + by + cz + d$ for an arbitrary oriented cutting plane. The cutting plane is then extracted by computing the isosurface of the new scalar field. In addition, another scalar function is mapped onto the cutting plane and interpolated along the isosurface.

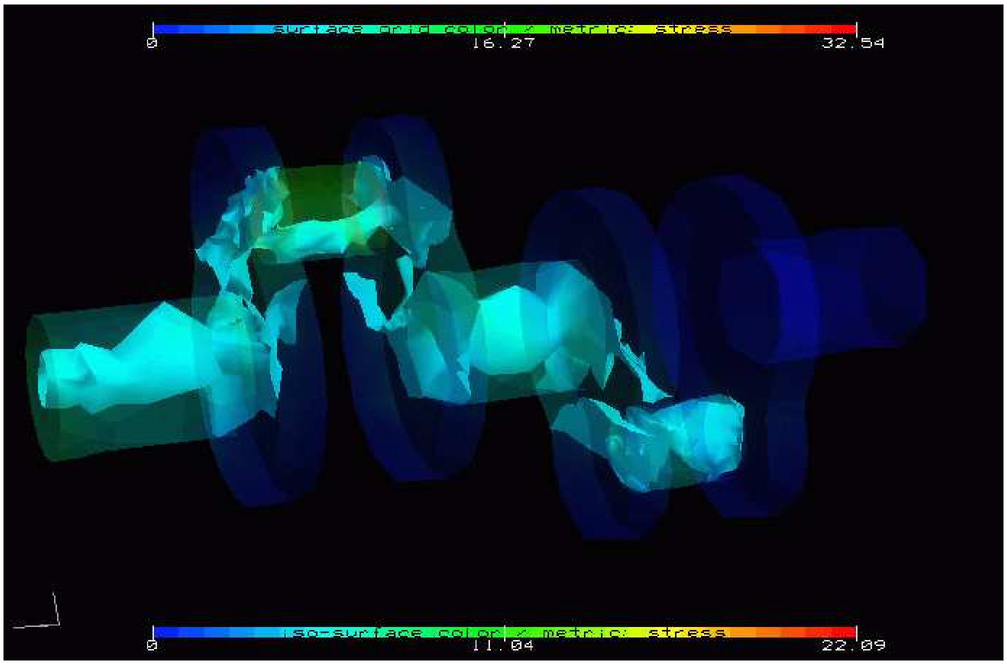

Figure 4: Example of an iso-surface

The ability to sweep a cutting plane back and forth in a 3D curvilinear grid is an important capability. It is generally not possible if the cutting plane spans a large grid because of the lack of CPU power. However, due to the applied multiresolutional approach our system is able to explore even large datasets composed of up to 800 000 tetrahedrons in this way. Figure 5 shows an example of a cut-plane.

5.5 Vector plots

Vector plots are the simplest global technique for visualizing vector fields and can be very effective. They consist of many small line segments or arrows which show the direction and magnitude at various points in the vector field.

Vector plots can be rendered with a variety of glyphs. The simplest representation is as a simple tuft (line segment). Arrow heads may be added to indicate the direction of the flow, and may be rendered as cones to take advantage of lighting and shading models. However, these added features come at a significant cost. For example, adding a simple 3D arrow head yields a three fold increase in the number of vectors. Displaying cones means calculating the vertices and normals for at least six polygons plus an added rendering cost.

5.5.1 Pitfalls

Vector plots drawn at grid vertices on structured or unstructured grids can produce misleading results if the grid is non-uniform. More arrows may be placed in areas where the grid density is large, or worse, the grid can cause destracting patterns in the output image. This is especially true in respect of FE++ due to its advanced adaptive numerical solution schemes which produce grids that exhibit great density variations.

For this reason we put a regular grid onto the unstructed grid of the cut plane. Thus the arrows are located at the vertex positions of the regular grid (see figure 6), which results in a significant visual improvement of flow perception. The interpolation of the data entries is done linearly because of its efficiency.

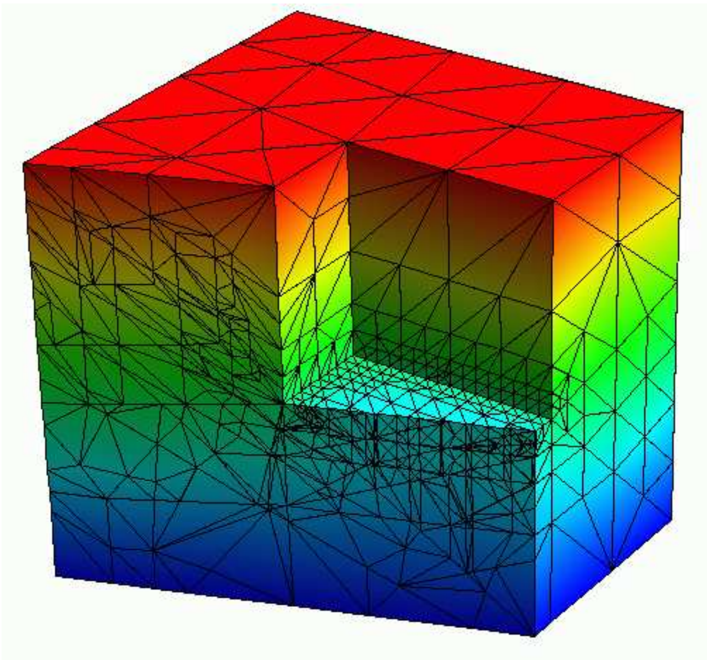

Figure 5: Example of a cut-plane

5.6 Element Shrinking

Shrinking the elements of an unstructured finite element grid is a convenient method to visualize the spatial structure of the elements that the grid consists of. Shrinking depictions are made by scaling the elements around its focal points (see figures 7 and 8).

5.6.1 Optimization

Due to the huge number of triangles to be rendered the method places extremely high requirements on the graphics hardware. First, the graphics subsystem has to offer a high fill rate to enable short rendering times. Second, it must provide a high floating point performance related to the geometry engines, since it must handle a large number of triangles, even if most of them are invisible to the user.

In order to reduce the load on the graphics subsystem we have assigned some geometry related transformation tasks to the CPU rather than to the graphics subsystem. In addition, we make use of back face culling in order to avoid rendering of faces not visible to the user.

To get an improved spatial impression of the grid structure it has proven useful to use biocular viewing devices, such as stereo glasses or head mounted stereo displays.

Further speed improvements may be expected from applying various culling methods, such as view frustum culling and occlusion culling [8]. Again, the main problem lies in the huge amount of triangles feed into the graphics pipeline, thought most of them will be invisible to the user, since they are covered by other triangles.

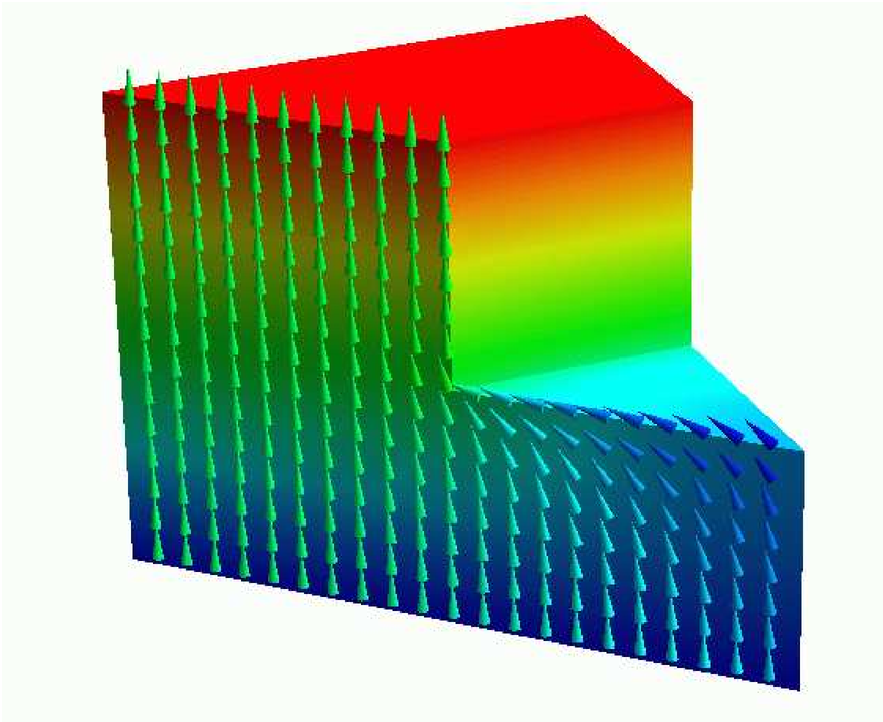

Figure 6: Example of a vector plot

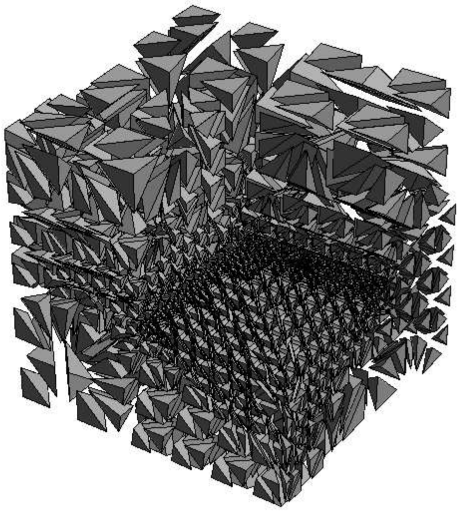

Figure 7: Example of a shinking depiction

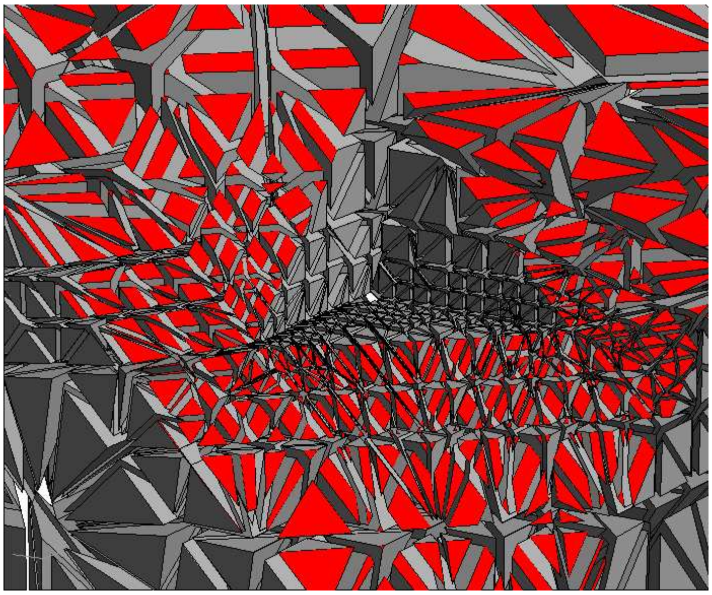

Figure 8: Example of a shinking depiction

5.7 Deformation Scaling

Deformation scaling is used to visualize model deformations that arise when applying forces to the model. In general, most deformations are invisible to the user because of their small quantities. For this reason the user is given the opportunity of specifying a scaling factor to scale-up invisible deformations (see figure 9).

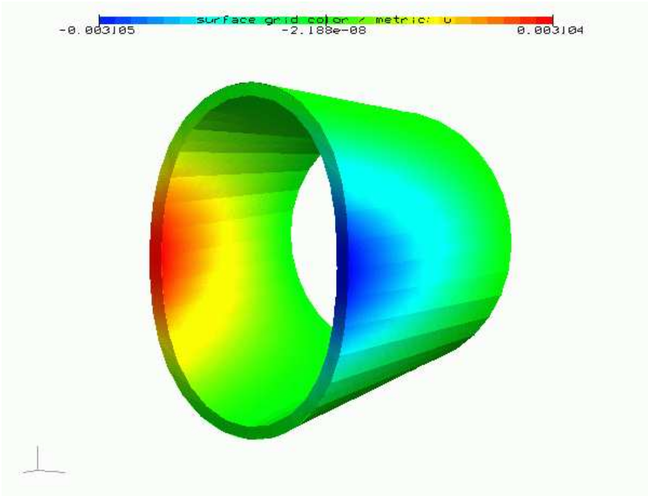

Figure 9: Example of deformation scaling

5.7.1 Pitfalls

Due to the linear scaled movement vectors, the user must take care of misleading results, which might differ from the real deformation to a notable degree (see figure 10). In particular deformations like torsion are prone to produce misleading depictions when scaling up the deformation vectors.

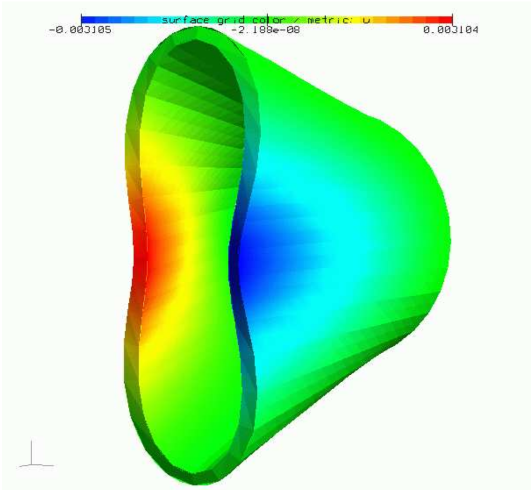

Figure 10: Example of wrong deformation scaling

6 Combining different visualization methods

The following picture (figure 11) demonstrates the capability of combining different visualization methods. Combining different visualization methods is a crucial feature of visualization systems to offer insights which provide maximum information to the user.

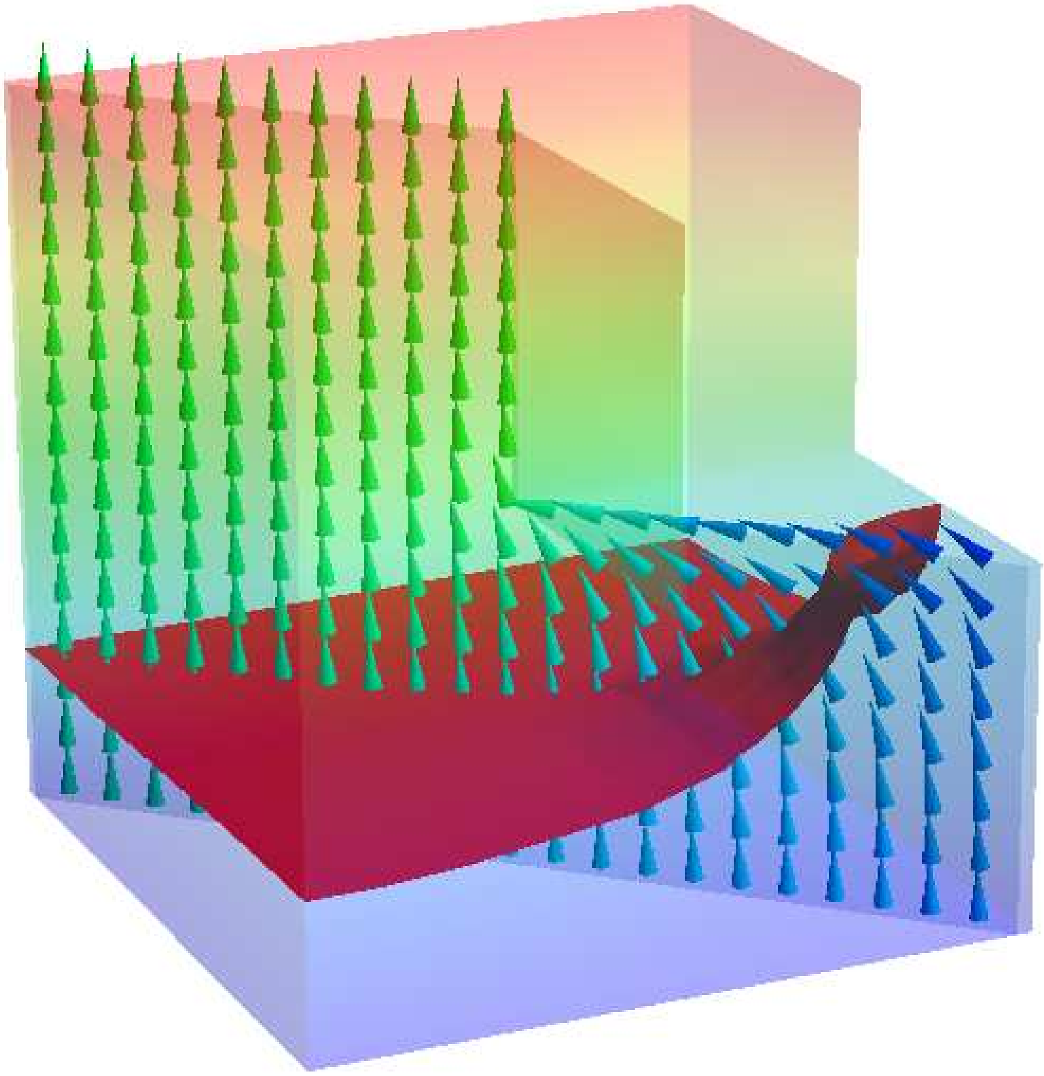

Figure 11: Combination of various visualization methods

References

[1] Joachim Schberl, ”FEPP – A Finite Element Package written in C++”, http://www.numa.uni-linz.ac.at/Staff/joachim/cpp/doc/index.html, 1999.

[2] Silicon Graphics, ”OpenGL Reference Manual”, Addison-Wesley Publishing Company, ISBN-0-201-63276-4, 1992.

[3] GRAPE User's Manual, Institut fr Angewandte Mathematik, Universitt Freiburg, Version 5.3, 1997.

[4] Information can be found at http://www.troll.no.

[5] David Kenwright, "Visualization Algorithms For Gigabyte Datasets", MRJ Technology Solutions, NASA Ames Research Center, M/S T27A-2, Moffett Field, CA 94035.

[6] Steve Bryson, Sandy Johan, "Time Management, Simultaneity and Time-Critical Computation in Interactive Unsteady Visualization Environments", MRJ, Inc. / NASA Ames Research Center.

[7] Michael Cox, David Ellsworth, "Managing Big Data for Scientific Visualization", MRJ/NASA Ames Research Center, Microcomputer Research Lab, Intel Corporation, 1997.

[8] Hansong Zhang, Dinesh Manocha, Tom Hudson, Kenneth E. Hoff III, "Visibility Culling using Hierarchical Occlusion Maps", in Proc. ACM SIGGRAPH, 1997.

[9] P. Cignoni, C. Montani, R. Scopigno, "A comparison of mesh simplification algorithms", Istituto di Elaborazione dell'Informazione- Consiglio Nazionale delle Ricerche, 1997.

[10] Jonathan Cohen, Dinesh Manocha, Marc Olano, "Simplifying Polygonal Models Using Successive Mappings", University of North Carolina at Chapel Hill.

[11] Amitabh Varshney, Pankaj K. Agarwal, Frederick P. Brooks Jr., William V. Wright, Hans Weber, "Generating Levels of Detail for Large-Scale Polygonal Models", Department of Computer Science, Duke University, North Carolina, CS-1995-20, 1995.