Efficient Visualization of Large–Scale Data on Hierarchical Meshes

Ralf Neubauer[1], Mario Ohlberger[2], Martin Rumpf[1], Ralph Schwörer[2]

[1] Institut für Angewandte Mathematik, Universität Bonn, Wegelerstr. 6, 53115 Bonn
[2] Institut für Angewandte Mathematik, Universität Freiburg, Hermann–Herder–Str. 10, 79104 Freiburg

Abstract. A multi-resolution approach is presented for data on a large class of hierarchical and nested grids. It is based on a procedural interface and a set of hierarchical and adaptive visualization methods. Such a method consists of a recursive traversal of mesh elements from the grid hierarchy combined with an adaptive stopping according to some error indicator which is closely related to the visual impression of data smoothness. During this traversal user data is only temporarily and locally addressed on single elements. No in advance mapping onto prescribed formats is necessary. The user only has to supply a set of element access routines as an interface to his specific data structures. As no extra storage is required, also large, economically stored computational grids can be handled on workstations with moderate local memory. Significant examples illustrate the applicability and efficiency on different types of meshes.

1 Introduction

Efficient numerical algorithms such as multi grid methods are nowadays capable to resolve complex structures in the simulation of physical processes. In a post processing step the user wants to explore the large amount of data with typically millions of unknowns interactively to improve the understanding of interesting features. Therefore efficient visualization tools are essential to extract the requested information from the enormous data base at a high frame rate.
The numerical methods are mostly based on a variety of domain discretizations such as structured or unstructured Finite Difference, Finite Element or Finite Volume grids, which are in general supplied with a hierarchical structure. These meshes may consist of a single or of mixed element types, e. g. simplicial, prismatic, rectangular or cuboidal, and they are frequently generated by different recursive, adaptive refinement strategies. Thereby non standard and application dependent data structures are often essential for an efficient implementation of the simulation algorithm.
The hierarchical type of these numerical data structures, first used for computing, is also well suited to improve the efficiency of a class of typical visualization methods. In the present paper two main aspects will be discussed.

First, we ask for a flexible integration of the above large class of hierarchical data structures from the applications into a post processing environment. The gap between the user's numerical data formats and the prescribed structures usually used by visualization tools is one of the fundamental outstanding problems in scientific visualization [7,15]. Most of the visualization software currently in use works on prescribed data formats [2,5,9,16]. User data has to be converted into such a format. But this is time and storage expensive especially in case of large nested grids, where very often closely related to the specific application an economical data storing is possible. It seems to be impossible to set up a fairly general and efficient data format covering all the above grid types. For non hierarchical meshes in [13] a different approach, which tries to avoid these difficulties, is proposed. A mesh is defined as a procedurally linked list of elements. There is no random access to a single element. Information about elements is only locally and temporarily provided by user supplied access procedures. This concept can be generalized to hierarchical grids. In section 2 we will introduce access routines to hierarchical elements supporting a recursive traversal of any nested grid hierarchy. Furthermore in section 3 a type of economical data structures to store nested grid data efficiently is discussed. We point out that such structures, which can exclusively been handled by a procedural approach, are well suited to store even very large grid geometries on a standard graphic workstation.

Second, especially the huge amount of data delivered by efficient numerical methods requires as well efficient visualization methods to support an interactive analysis of the physical characteristics modeled by the simulation. We will discuss mainly the extraction of isosurfaces as a typical graphical tool to inspect 3D data sets, although this methodology applies to other methods as well. The classical marching cube method, by Lorensen and Cline [11], efficiently renders local isosurfaces on hexahedral grids. But the underlying search algorithm for intersected elements is still an overall traversal of the set of elements. Mainly three different types of improvements have been investigated: an efficient presorting of elements [3,10,14], a seed point selection strategy combined with a spreading search for isosurfaces using adjacency information [6], and the recursive hierarchical search for intersected elements using precomputed and stored min/max values [17]. We sketch here how to combine a hierarchical search over the grid hierarchy with an adaptive stopping on elements where the visual improvement, one would obtain on finer grid levels, is below a user–prescribed error tolerance. Such an adaptive stopping on coarser grid levels has been considered for instance in the context of volume rendering for an adaptive splatting technique [8] and for an algorithm based on successively refined tetrahedral Delaunay meshes [1]. The advantage of our algorithm is, that it can be implemented on a fairly general class of hierarchical grids, which are in our case procedurally addressed, acts strictly local on elements without referring to their neighbourhood, and rules out discontinuities on the isosurface due to transitions between different grid levels. This concept can easily been generalized to other visualization methods, such as color shading or isoline drawing in 2D, or on volume slices in 3D. It applies to hierarchical meshes consisting of elements, which are tensor products of simplices in one, two or three dimensions and the corresponding function spaces generated by tensor products of linear functions on simplices. This especially includes the elements sketched in Fig. 1 with, for instance, linear, bilinear or trilinear functions defined on them. A detailed discussion of this concept can be found in [12]. Section 4 gives a brief overview on the basic ideas and in section 5 several examples underline the applicability of the presented concept. Let us finally clarify some

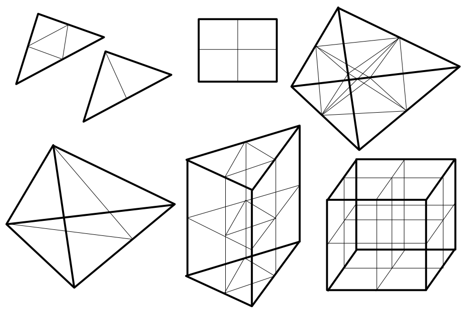


Fig. 1. The basic element types and their refinement (for triangles and tetrahedrons we depict two types of refinement).

notation concerning the grids, which will be used throughout the subsequent sections. A set of nested grids $\{\mathcal{M}^l\}_{0\le l\le l_{\max}}$ is a family of meshes, which are recursively generated by refinement of certain elements of the preceding coarser mesh. More explicit, an element $E \in \mathcal{M}^l$ is refined according to some refinement rule and we thereby obtain a set of child elements $\mathcal{C}(E) \subset M^{l+1}$ (cf. Fig. 1). By refining an element new vertices x are generated. Let us denote the set of vertices of E and $\mathcal{C}(E)$ by $\mathcal{N}(E)$, $\mathcal{N}(\mathcal{C}(E))$ respectively.
We finally remark that, although we mainly focus on the 3D case, most of the schematic figures deal with the 2D case, to simplify the presentation.

2 A procedural interface to hierarchical meshes

In [13] a visualization interface for arbitrary meshes with general data functions on them has been proposed. This interface tries to avoid restrictions on the element types. A mesh is defined as a procedurally linked list of non intersecting elements. The access to data is done by user supplied procedures addressing the user data structures and returning the required data temporarily in a prescribed element structure *ELEMENT*. It especially contains a reference to some element type, the coordinate vectors for the nodes, and function data on them. (Here we restrict ourselves to the basic concept. The true data structure is slightly more general, especially concerning the interface for function data). In general at the same time only one *ELEMENT* structure is present in storage. There is no random access to a single element. But this is in fact not necessary for most

common and frequently used visualization methods (cf. section 4). Especially no permanent mapping of numerical data onto new data structures is required. The visualization tools directly work on the data structures the user is accustomed to from his numerical method. He only has to provide the access procedures and give a description of the element types. For the details on the procedural interface we refer to [13].
So far no hierarchical structure is taken into account. Now we enlarge this interface by access procedures which procedurally represent the tree structure underlying a nested grid. Two procedures *first_macro()*, *next_macro()* successively deliver information on the coarse grid elements in an *ELEMENT* data structure, overwriting previous element data. A call of *first_child()* generates and fills an additional *ELEMENT* structure with some first child data. Finally successive calls of *next_child()* traverses the other child elements of the same parent element and replaces previous child data (cf. Fig. 2). Thereby during a recursive traversal of the grid hierarchy, a list of at most n temporarily filled ELEMENT structures is present in memory at the same time, where n is the depth of the hierarchy. This especially implies that also in the user data structures the element information needs not to be stored completely on all grid levels, but it may be generated when needed, based on complete parent information and economically stored offset data (cf. section 3).

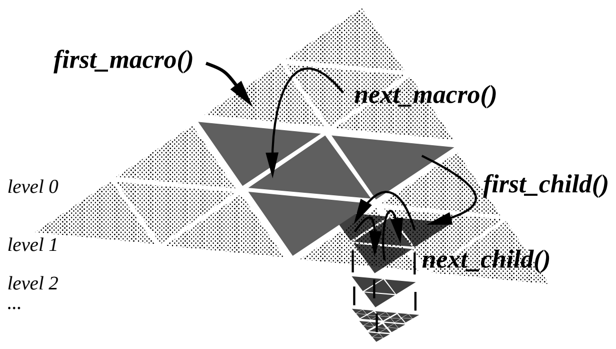


Fig. 2. A schematic sketch of the procedural access to hierarchical grids by the four routines *first_macro(), next_macro(), first_child(), next_child().*

The above access procedures supply a visualization method with all necessary information to locally evaluate and graphically represent grid geometry and data. This is sufficient to run merely all visualization algorithms, e. g. isosurface rendering, isoline drawing and data dependent color shading. In section 4 we will see that additional error indicator values on vertices are required by the hierarchical and adaptive visualization methods. Furthermore we need upper and lower bounds for data on elements, typically computable based on the error indicator values. Additional procedures are included to supply visualization methods with such information.

Particle tracing and related methods can be implemented similarly. For an efficient implementation we here have to process elements in the order in which they are traversed by an integral line of some vector field, e. g. a particle path. If we run the integration on level i, we therefore have to recover the "hierarchical history" of an adjacent i level element in each step, which can be done recursively. Here the hierarchical history is described as the corresponding list of *ELEMENT* structures and the recovering is done adjusting the list corresponding to the current element. This requires a procedure *neighbour()* on the coarse grid level and an evaluation of adjacency relations among the set of children of any parent element (cf. section 3).

3 Economically stored nested grids

One of the main advantages of a procedural access to hierarchical data is that economically stored hierarchical grids can be addressed directly by the visualization. This allows us to handle even very large grid geometries interactively on a graphics workstation. Let us now sketch the minimal information, which is to be stored to address a hierarchical grid procedurally for numerical or graphical purposes.
For each element the geometry of its children is uniquely described by the refinement rule, if no additional grid alignment is taken into account. I. e. the coordinates of each $l+1$ level vertex x^{l+1} in $\mathcal{N}(\mathcal{C}(E)) \setminus \mathcal{N}(E)$ can be evaluated as weighted sums over the coordinates of the l level parent vertices $x^l \in \mathcal{N}(E)$ with weights $\omega_{x^{l+1}}(x^l)$ depending solely on the refinement rule:

$$x^{l+1} = \sum_{x^l \in \mathcal{N}(E)} \omega_{x^{l+1}}(x^l)\, x^l$$

In general the number of refinement rules is small, such that element and vertex production rules, including the weights, can easily be stored in a lookup table. On curved boundary segments, where vertices generated by the refinement rules on the grid boundary are pushed afterwards onto the continuous smooth boundary, these new coordinates have to be stored additionally. We skip a detailed discussion here and only remark that the storage requirement is typically of lower order, because the boundary is a lower dimensional set.
A general hierarchical grid can be described by a list of macro elements, and the subdivision history for each of them, given by a tree of hierarchical elements. Therefore, in the user's application a macro element data structure *MacroElement* contains full information on the corresponding element: an identification of the nodes, including their coordinates or a reference to them, adjacency relations across element faces, and a reference to the tree of child elements. An economical hierarchical element data structure *EcoElement*, which corresponds to a node in any macro element's subdivision tree, consists of the index of the element's refinement rule, an array of identifiers for the new nodes generated by the refinement, to address data values on them, and finally a reference to an array of child elements (cf. Fig. 3). Let us mention that the highest level elements

```
MacroElement
MacroNode       *macroNode
MacroElement    *neighbour
EcoElement      *self
```

```
EcoElement
int             refrule
Node            **newNode
BoundPoint      *boundPoint
EcoElement      *child
```

Fig. 3. A sketch of possible user data structures which fully describe a hierarchical grid in a pseudo C notation (MacroNode and Node are the reference types for vertices on the coarse grid and new vertices generated during the refinement respectively. BoundPoint is the structure to store vertices on curved boundary segments)

are not represented explicitly in the data base. Complete information on those elements is already present on their parent's level.

A typical visualization method now runs over the list of macro elements, and recursively processes higher level elements of the hierarchical tree. Thereby economically stored information for a child element is temporarily completed referring to the economical data structure for hierarchical elements, given as *EcoElement* data, and complete parent element information, which has already been stored in an *ELEMENT* structure (possibly slightly enlarged by some additional user data) in a preceding step of the recursion. Let us mention that besides the nodal coordinate vectors and the data references we can also generate information on element adjacency recursively, where we identify neighbouring cells of child elements as child elements of neighbouring cells.

Let us finally estimate storage requirements and thereby capabilities of a procedural approach, where we assume that only one refinement rule is used, which generates c children in each step. Now consider n levels of global refinement on a macro grid with m elements. Then the required storage for the hierarchy is

$$m\left(\frac{c^n-1}{c-1}\cdot EE+ME\right)$$

where EE, ME is the storage requirement for a single *EcoElement* or *MacroElement* structure respectively and furthermore $\frac{EE}{c-1}$ is an estimate for the storage needed per element on the finest grid level, if we neglect the small and constant memory block for the macro elements. A hierarchical, tetrahedral grid consisting of 10 million tetrahedrons on the finest level, where tetrahedrons are divided into eight children in each refinement step, with six new nodes per refinement (cf. Fig. 1), can be stored in $\sim$ 46 MB (here we suppose a need of 4 bytes in storage per integer, floating point number, and pointer). Compared to this, a non hierarchical storing of the finest grid level would require at least $\sim$ 340 MB (4 nodes and 4 adjacency references, plus $\sim \frac{1}{6}$ coordinate vector per element) without the chance to run hierarchical visualization methods.

4 An hierarchical and adaptive visualization strategy

Visualization methods, especially on 3D data sets such as isosurface extraction and color shading on slices, can benefit from the nested structure of the underlying grid. In the following we will focus on the isosurface case. Similar considerations hold for other visualization methods as well.
The cost to extract an isosurface from a given volume data set can be reduced enormously, taking into account hierarchical information. Instead of traversing all elements, like a standard marching cube strategy does, we can recursively test for intersections on coarser level elements E to decide whether the children $\mathcal{C}(E)$ have to be visited or not. If the considered function on the grid is smooth, this leads to a cost reduction of one order of magnitude up to a logarithmic factor. The intersection test on an element requires the calculation of robust data bounds. Simply taking into account the function values on the element vertices $x^l \in \mathcal{N}(E)$ will not be sufficient, because we might overlook information apparent on finer grid levels only, e. g. strongly curved segments of an isosurface (cf. Fig. 4).
Let us suppose that we have at hand an estimate for the function's second

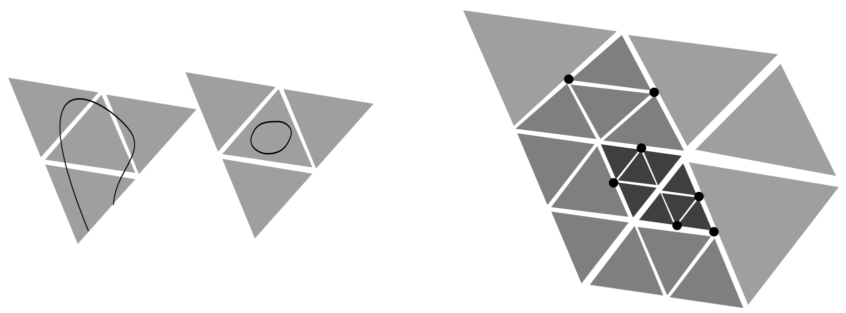


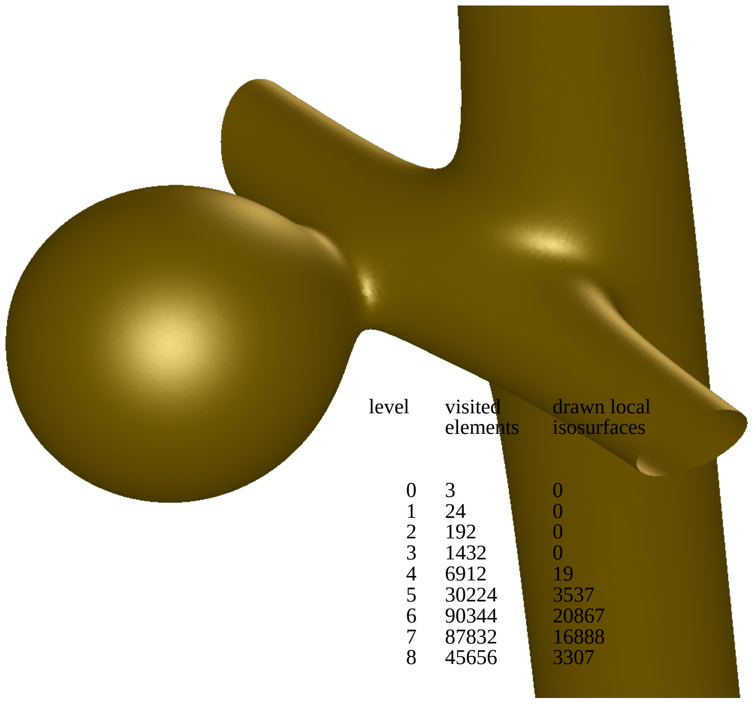
Fig. 4. On the left: isoline segments in 2D will be missed if just vertex information is taken into account to test for intersections (The same holds in 3D for isosurfaces), on the right: An adaptive traversal of a 2D grid leads to hanging nodes.

derivatives — or something comparable — on each element of the hierarchy. Then a straightforward calculation of bounds is possible by means of Taylor expansion. In what follows we will see that such quantities are also essential to support an adaptive visualization strategy and can therefore with some care be reutilized here. For details we refer to [12].
Up to now, full information on the finest grid level is always extracted and visualized, disregarding that in areas where the isosurface is smooth, a considerably coarser resolution would visually be acceptable as well. But this would restrict the method's tree traversal to a coarser subtree and thereby considerably reduce the amount of graphic primitives which have to be rendered, and thereby substantially speed up image generation without missing fine details in other areas of the data sets. The locally coarser resolution can be obtained by an adap-

tive stopping during the recursive traversal of the grid hierarchy on elements E, where some error indicator $\eta(E)$ is below a given threshold value ϵ. There is a variety of possible error indicators related to the local smoothness of the data (cf. [12] for a comparison). We found the jump of the normalized function gradient on the element faces to be an appropriate error criteria, because it measures angles between adjacent polygons, locally representing an isosurface. These jumps can be calculated and stored in a precomputing step on the vertices of the hierarchical grid, lying on the corresponding element faces. The error criteria $\eta(E)$ on an element E is then defined as the maximal indicator value on all its faces, i. e. on the vertices $x^{l+1} \in \mathcal{N}(\mathcal{C}(E)) \setminus \mathcal{N}(E)$. Following the adaptive strategy, hanging nodes will in general be unavoidable. They occur on faces where we have a transition from coarser to finer elements, which are traversed by our adaptive algorithm (cf. Fig. 4). Hanging nodes lead to cracks in the isosurface. To rule out these artifacts we have to replace the true function value v on the finer elements at the considered face by interpolated values Iv coinciding with the function on the adjacent coarser element. The decision, whether to take the original function value, or the interpolation at a specific vertex, has to be based on the error indicators as well, i. e. we choose the interpolation at a vertex, if the stored indicator value is larger than ϵ. The interpolation is evaluated analogously to the calculation of coordinates for child vertices (see section 3). If we suppose the indicator values on coarser level vertices to be larger than indicator values on vertices appearing on the children of the corresponding elements, the resulting interpolation will be continuous and cracks will be ruled out (for bilinear faces some additional considerations have to be taken into account [12]). This is a natural assumption especially for fine grid levels and smooth data. In case of violations we adjust the indicator values, according to the condition, in an additional precomputing step. Let us emphazise that we operate only locally on single elements and thereby avoid an expensive non local construction of a conforming closure. The following algorithm sketches the hierarchical and adaptive isosurface extraction for an isosurface value α :

```
AdaptInspect($\alpha$,$E$) {
    $Iv$ = Interpol($v$, $E$);
    if IntersectionTest($\alpha$,$Iv$,$E$) {
        if $(\mathcal{C}(E) \neq \emptyset) \wedge (\eta(E) > \epsilon)$
            for all $\tilde{E} \in \mathcal{C}(E)$
                AdaptInspect($\tilde{E}$);
        else Extract($\alpha$,$E$);
    }
}
```

where *Interpol()* is the implementation of the above interpolation operator, *IntersectionTest()* checks whether α is contained in the image interval of the currently considered local function, and *Extract()* renders the local intersection of the isosurface with an element. Let us finally mention that our error indicator $\eta(E)$ can be utilized similar to a bound of the second derivatives to estimate



level	visited elements	drawn local isosurfaces
0	3	0
1	24	0
2	192	0
3	1432	0
4	6912	19
5	30224	3537
6	90344	20867
7	87832	16888
8	45656	3307

Fig. 5. An adaptive isosurface extracted from a 50 331 648 element data set. A table lists for each grid level the number of visited cells and the number of drawn local isosurfaces due to the adaptive stopping criteria.

higher order data contributions on a specific element. Thereby it allows a robust implementation of the intersection test without referring to additional information.

5 Examples and Applications

To illustrate the performance of the hierarchical and adaptive strategy for different applications we now discuss some test cases. Fig. 5 depicts a significant example of an adaptive isosurface on a test data set which consists of precalculated values of an analytic function at the vertices of a tetrahedral grid. Fig. 6 deals with the density from a porous media calculation based on a hexahedral mesh (Numerical data provided by S. Oswald, Zürich). The hexahedrons are successively bisected, every time in one direction, cycling over x, y and z. The data is timedependent. Above, the isosurface for a fixed value is drawn at different times. They especially enlighten the adaptive strategy. Below a color shading of the density on an intersection plane is shown at the same times. Again black lines mark the intersected element faces. Function data at any time is interpolated based on a small set of time steps equipped with the corresponding data.

Thereby the maximal error indicator of two time steps can be utilized as a possible error indicator for any interpolated data in between. This distinguishes our approach from the other efficient non hierarchical isosurface methods where a preroll is unavoidable for each new time. In Fig. 7 we analyze the behaviour of the hierarchical and adaptive approach for an isosurface with a cusp type singularity on a sequence of successively refined tetrahedral grids. They are economically stored as described in section 3 and the final grid consists of ~ 12 million elements. At each refinement step a tetrahedron is divided into eight child elements. A diagram shows in a logarithmic scale the total number of visited elements for different grid levels, including the coarser level elements passed by the hierarchical algorithm. We compare the results for a standard marching cube type algorithm, where we only count the elements on the finest level, with

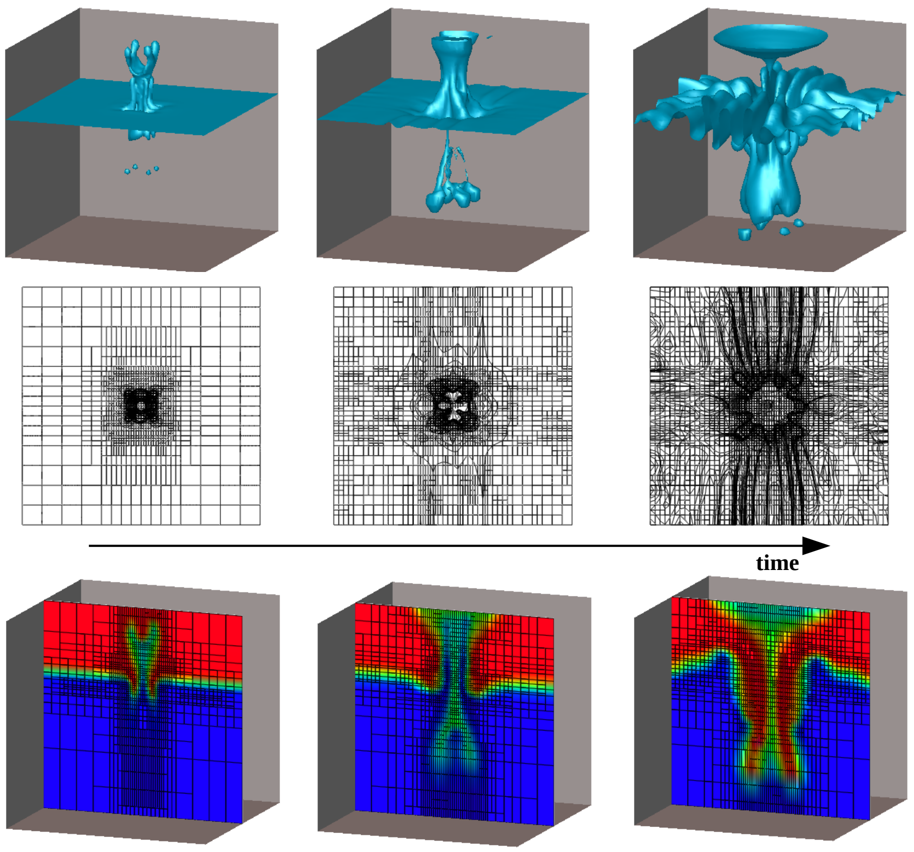


Fig. 6. At different times porous media data is visualized using isosurfaces (above) and color shading on slices (below). Additional black lines mark intersections with element faces in a projective view (in the middle) corresponding to the isosurfaces and on the intersection plane itself (below).

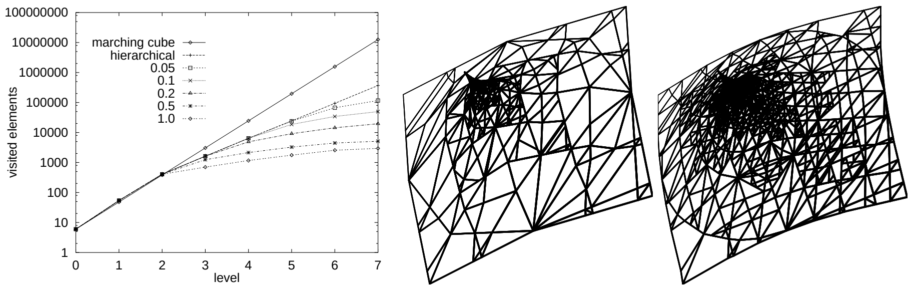


Fig. 7. In a diagram on the left we compare the proposed isosurface method for different threshold values ϵ with the standard marching cube type algorithm. On the right for two different ϵ grid models of the extracted isosurfaces are drawn.

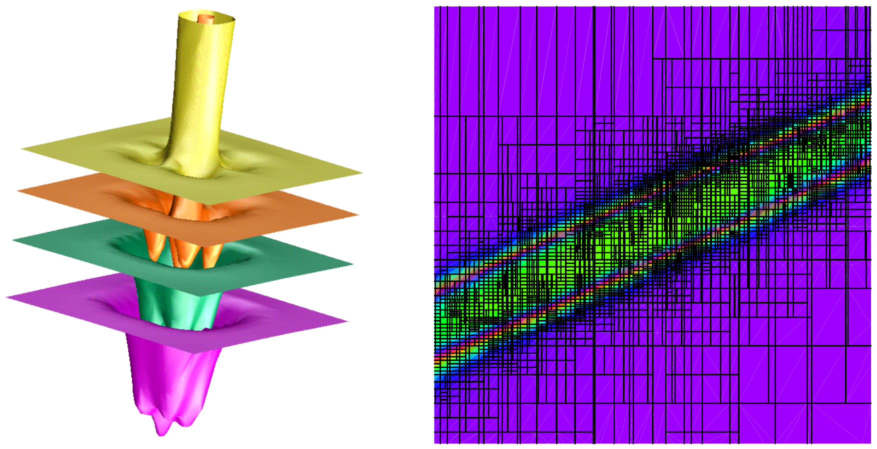


Fig. 8. Several adaptive isosurfaces from a porous media data set and and an adaptive slicing of a non–conforming hexahedral grid.

those for a solely hierarchical and for several hierarchical and adaptive runs of the algorithm, corresponding to different threshold values ϵ. This reflects the theoretically expected costs and underlines the considerable data reduction capability of the adaptive approach. Finally Fig. 8 shows a combination of several smoothly shaded, adaptive isosurfaces from the same data set and the result of an adaptive slicing algorithm on a non–conforming adaptive hexahedral grid on which a discrete solution of the Hamilton–Jacobi–Bellman equation is given (Numerical data provided by L. Grüne [4], Augsburg). It demonstrates the applicability of our procedural approach to non standard grid geometries.

Acknowledgement

The authors thank W. Kinzelbach and S. Oswald from Zürich, who provided the porous media data set and L. Grüne from Augsburg for the non–conforming adaptive data set.

References

1. Cignoni, P., De Floriani, L., Montani, C., Puppo, E., Scopigno, R.: Multiresolution Modeling and Visualization of Volume Data based on Simplicial Complexes, Proceedings of the Visualization'95 , 19-26, 1995.
2. Dyer, D. S.: A dataflow toolkit for visualization, IEEE CG&A 10, No. 4, 60–69, 1990
3. Giles, M.; Haimes, R.: Advanced interactive visualization for CFD, Computing Systems in Engineering, 1(10): 51-62, 1990.
4. Grüne, L.: An adaptive grid scheme for the Hamilton-Jacobi-Bellman Equation, to appear in Numer. Math.
5. Haber, R. B.; Lucas, B.; Collins, N.: A data model for scientific visualization with provisions for regular and irregular grids, Proc. IEEE Visualization '91
6. Itoh, T.; Koyamada, K.: Isosurface Generation by Using Extrema Graphs, 77-83, 1994.
7. Lang, U.; Lang, R.; Rühle, R.: Integration of visualization and scientific calculation in a software system, Proc. IEEE Visualization '91
8. Laur, D.; Hanrahan, P.: Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering, ACM Computer Graphics 25 (4), 285-288, 1991.
9. Lucas, B.; et. al. : An architecture for a scientific visualization system, Proc. IEEE Visualization '92
10. Livnat, Y.; Shen, H. W.; Johnson, C. R.: A near optimal isosurface extraction algorithm using the span space, Transaction on Visualization and Computer Graphics, 2 (1), 73-83, 1996.
11. Lorensen, W.E.; Cline, H.E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm, ACM Computer Graphics 21 (4), 163-169, 1987.
12. Ohlberger, M.; Rumpf, M.: Hierarchical and adaptive visualization on nested grids, Mathematische Fakultät, Universität Freiburg, Preprint 22, 1996, to appear in Computing
13. Rumpf, M.; Schmidt, A.; Siebert, K. G.: Functions defining arbitrary meshes, a flexible interface between numerical data and visualization routines, Computer Graphics Forum 15 (2), 129-141, 1996.
14. Shen, H.-W.; Johnson, C.R.: Sweeping Simplices: A fast iso-surface extraction algorithm for unstructured grids, Proceedings of the Visualization'95 , 143-150, 1995.
15. Treinish, L. A.: Data structures and access software for scientific visualization, Computer Graphics 25, 104–118, 1991
16. Upson, C.; et. al.: The Application Visualization System: A computational environment for scientific visualization, IEEE CG&A 9, No. 4, 30–42, 1989
17. Wilhelms, J.;van Gelder, A.: Octrees for faster isosurface generation, ACM Trans. Graph. 11 (3), 201–227, 1992.