

# A Procedural Interface for Multiresolutional Visualization of General Numerical Data

T. Geßner      B. Haasdonk      R. Kende      M. Lenz      M. Metscher      R. Neubauer  
M. Ohlberger      W. Rosenbaum      M. Rumpf      R. Schwörer      M. Spielberg  
U. Weikard

18th August 1999

## Abstract

*Together with a rapid development of computer hardware, sophisticated, efficient numerical algorithms allow simulation computations of complex physical phenomena. Methods, such as Finite Volume, Multigrid Finite Element schemes, Sparse Grid, Wavelet approaches, and Particle Methods or Gridless Discretizations all carry their own, tailored data structures, which reflect the decomposition of the function spaces as well as the decomposition in physical space.*

*Multiresolutional visualization on numerical data is described as an indispensable ingredient of real time interactive post processing. The typically enormous data bases are locally resolved on different levels of detail to achieve a significant saving of CPU and rendering time.*

*For efficient data analysis and graphical post processing the method of spatial, hierarchical subdivision combined with the recovery of the local function spaces is presented. To manage a variety of different numerical data a general procedural interface to arbitrary large numerical data sets is presented. This leads to a visualization beyond prescribed data formats. Discrete numerical solution data is directly addressed in the user's data structures. Furthermore the procedural interface supports a flexible method of local error measurement, again encapsulated in certain user supplied functions. The software conception, its data classes and methods are described and the setup of the corresponding procedural user interfaces is discussed in detail.*

*Examples from various numerical methods and different data bases underline the applicability of the proposed concept.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>Part I Abstract Concepts</b>		<b>6</b>
<b>2</b>	<b>Grid hierarchy and hierarchical function bases</b>	<b>7</b>
2.1	Finite Volume Methods . . . . .	8
2.2	Multilinear Finite Elements . . . . .	9
2.3	Higher order polynomial Finite Elements . . . . .	9
2.4	Sparse Grids . . . . .	10
2.5	Wavelets and Pre-Wavelets . . . . .	10
2.6	Gridless Discretizations, Particle Methods . . . . .	11
<b>3</b>	<b>Visualization by hierarchical spatial subdivision</b>	<b>12</b>
3.1	Finite Volume Methods . . . . .	15
3.2	Multilinear Finite Elements . . . . .	16
3.3	Higher order polynomial Finite Elements . . . . .	16
3.4	Sparse Grids . . . . .	18
3.5	Gridless Discretizations, Particle Methods . . . . .	18
<b>4</b>	<b>Projection criteria and error indicators</b>	<b>18</b>
<b>5</b>	<b>Timedependent Data</b>	<b>21</b>
<b>6</b>	<b>Further Aspects on the Adaptive Strategy</b>	<b>23</b>
<b>7</b>	<b>Efficient Hierarchical Searching on arbitrary Nested Grids</b>	<b>24</b>
<b>8</b>	<b>Concept for the Procedural Data Access</b>	<b>25</b>
<b>9</b>	<b>Handling Ecomonically Stored Very Large Data</b>	<b>26</b>

<b>Part II Implementational Aspects</b>	<b>29</b>
<b>10 Description of the Interface Structures</b>	<b>30</b>
<b>11 Writing Display Methods for the Classes HMesh2d and HMesh3d</b>	<b>38</b>
11.1 Introduction . . . . .	38
11.2 Preparation . . . . .	39
11.3 Data Extraction . . . . .	40
<b>12 Hierarchical Searching using the Interface</b>	<b>46</b>
<b>13 An Interface Recipe</b>	<b>48</b>
13.1 Introduction . . . . .	48
13.2 The Bismesh Structure . . . . .	50
13.3 Data Structures of the Interface . . . . .	51
13.4 get_new_hmesh() . . . . .	52
13.5 Routines to Traverse the hmesh . . . . .	54
13.6 Handling INTERFACE_ELEMENTS . . . . .	58
13.7 Helement Description . . . . .	62
13.8 Geometry Error Estimator . . . . .	66
13.9 Routines for Functions on hmeshes . . . . .	67
<b>14 Fundamental Methods on Hierarchical Meshes</b>	<b>71</b>
14.1 GenMesh . . . . .	71
14.2 Mesh2d . . . . .	73
14.3 HMesh2d . . . . .	77
14.4 Projects for Meshes . . . . .	84
14.5 Methods on other classes related to meshes . . . . .	85
14.6 HMesh3d . . . . .	86

## 1 Introduction

For the efficient simulation of physical phenomena various types of numerical methods have been developed. We mention here especially Finite Volume [23], Finite Element [5], Sparse Grid [47, 12], Particle [30], and Gridless Discretization schemes. They all approximate the continuous simulation solution in some discrete function space. Numerical efficiency is in general obtained by an intrinsic hierarchical structure in the data representation, which for instance allows the application of a multigrid strategy [16, 18]. In many applications the concrete hierarchical function space is closely related to some hierarchical, spatial decomposition. This also enables a multiresolutional approach in the post processing, where visualization methods recursively traverse the hierarchy in space, typically a nested grid, and evaluate the local, discrete solution.

Unfortunately there is no unique data representation shared by all these methods. Frequently non standard and application dependent data structures are often essential for an efficient implementation of the simulation algorithm. This 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 [24, 42]. Most of the standard visualization software currently in use works on prescribed data formats [7, 15, 27, 44].

Nevertheless in the general case a unique design of the visualization software is possible, if we encapsulate the actual spatial grid hierarchy and the access to the local function space in a procedural interface which has to be provided by the user. For the case of non hierarchical Finite Element data such an interface has been presented in detail in [34, 33]. Well known post processing methodology, which is based on this interface, supports the user with efficient tools. Thereby they directly address the user's numerical data structures.

Concerning the multiresolutional analysis based on hierarchical data we at first present an abstract frame, the *method of hierarchical spatial subdivision and adaptive data projection*, which applies to any type of numerical data and error measurement. It restricts to a depth first traversal of the corresponding grid hierarchy in case of visualization methods such as isoline and isosurface extraction, volume rendering, displaying graphs and colorshading in 2D or on slices in 3D. In case of particle tracing type methods highly efficient hierarchical searching is applied. In any case computationally expensive handling of adjacency relations is not required. On this background we are able to introduce the software concepts for a unique and general access to general hierarchical data.

A variety of different multiresolutional visualization strategies has been proposed in the literature. The following brief review is far from being complete. The simplification of triangulated surfaces [43, 17, 36] and their progressive transmission [20] is now already a classical topic in visualization. Adaptive volume rendering [25], the hierarchical [45] and the adaptive extraction of isosurfaces [39] have been discussed on octtree encoded data. For tetrahedral grids multiresolutional rendering has been presented in the context of Delaunay meshes, which generate a triangulation for an arbitrary set of sample points [6] and for retriangulations of data originally given on structured grids [14]. Different types of error measurement can be applied. The  $L^2$ , respectivly the  $H^{1,2}$  type error is taken into account in [25, 14],  $L^\infty$  errors are the foundation of the algorithm in [20, 39], the Hausdorff distance between surfaces is considered in [22], whereas wavelet coefficients serve as error criteria in [13, 8, 35]. Our concept is strongly influenced by its background of numerical adaptive multilevel methods. Furthemore it strictly separates mathematical data analysis from algorithmical aspects. The method was first developed for multilinear Finite Element data in [31, 29] and has been conceptionally enlarged in [32]. Here we discuss a further generalization to arbitrary nested function spaces. Its concrete implementation is efficient, if an appropriate choice of the spatial hierarchy is combined with an efficient retrieval of function data on the elements. For different types of numerical data the main aspects of the resulting algorithm are discussed. Examples indicate the range of applicability of this approach. The adaptive hierarchical visualization is based on local error measurement. Thereby the corresponding error indicators are defined on the hierarchical base functions. It is explained how this leads to a stopping criterion on grid cells during the recursive traversal without destroying the continuity of the obtained adaptive projection.

The main intention of this paper is the detailed discussion of an effective and efficient algorithmical base for the procedural access and the essential access routines and data structures, which have to be filled only locally and temporarily.

It is therefore divided in two main parts:

- Abstract Concepts
- Implementational Aspects

The first introduces the concepts in an abstract form and presents the general frame, whereas the second deals with the implementation based on a procedural interface, corresponding classes and methods.

## **Part I      Abstract Concepts**

## 2 Grid hierarchy and hierarchical function bases

In the following section we will discuss fairly arbitrary, hierarchical data to clarify the range of desired applicability of our post processing methods. The most important point is that there is in general no uniform relation between the hierarchical structure relevant in the function space, in which the simulation data resides, and a hierarchical spatial decomposition of the physical space, on which post processing methods are typically based. Therefore we at first briefly introduce separate hierarchies for both and then discuss their relations. In separate paragraphs we analyse this for a variety of important concrete cases. We start with a brief overview on nested spatial subdivision. Let  $\Sigma^n \subset \mathbb{R}^n$  be the set of simplices of dimension  $n$ , e. g.  $\Sigma^2$  the set of triangles and  $\Sigma^3$  the set of tetrahedrons. We consider all elements  $E \subset \mathbb{R}^n$ , the building blocks of the spatial hierarchy, which consist of tensor products of simplices, i. e.  $E = \bigotimes_{i=1}^k \sigma_i$  with  $\sigma_i$  in some  $\Sigma^m$ .

For example triangles and rectangles in 2D and tetrahedrons, prisms or hexahedrons in 3D are tensor product elements (cf. Fig. 1).

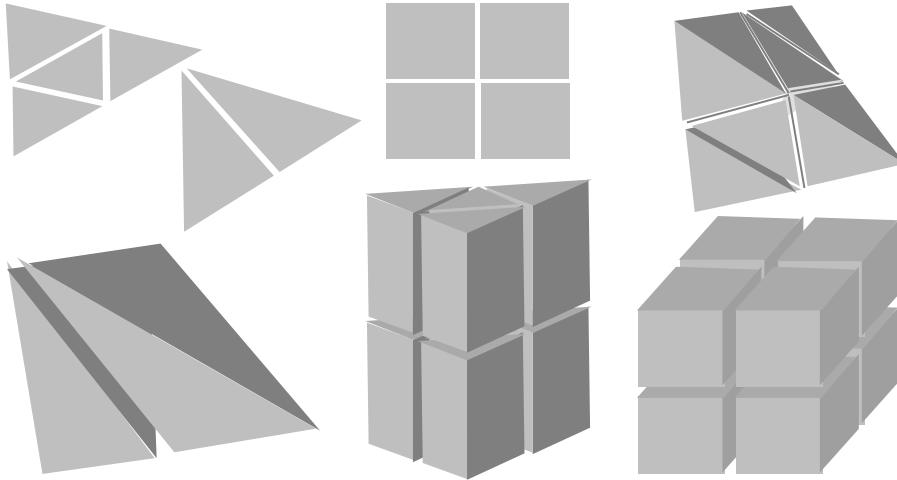


Figure 1: Basic element types in two and three dimensions with possible refinements.

A conforming mesh  $\mathcal{M}$  for a domain  $\Omega \subset \mathbb{R}^n$  is a set of closed elements  $E$  such that  $\bigcup_{E \in \mathcal{M}} E = \Omega$  and any two elements of  $\mathcal{M}$  are disjoint or they intersect in a boundary simplex, e. g. a common face, edge or vertex. A family of conforming meshes  $\{\mathcal{M}^l\}_{0 \leq l \leq l_{\max}}$  is called a nested grid, if for all  $E^{l+1} \in \mathcal{M}^{l+1}$  there exists an  $E^l \in \mathcal{M}^l$  with  $E^{l+1} \cap E^l = E^{l+1}$ . We suppose these grids to be recursively generated by a finite set of refinement rules applied to certain elements of the preceding, coarser mesh. Those elements  $E^{l+1} \in \mathcal{M}^{l+1}$  generated from  $E^l \in \mathcal{M}^l$  by subdivision will be called child elements  $\mathcal{C}(E^l)$  of element  $E^l$ . Vice versa the parent relation  $\mathcal{P}(E^{l+1}) := E^l$  holds for every  $E^{l+1} \in \mathcal{C}(E^l)$ .

On the other hand we assume a family of discrete (finite dimensional) function spaces  $\{\mathcal{V}^m\}_{0 \leq m \leq m_{\max}}$  to be given. These spaces are supposed to be nested, i. e.

$$\mathcal{V}^0 \subset \mathcal{V}^1 \subset \cdots \subset \mathcal{V}^{m_{\max}}.$$

A solution function  $u : \Omega \rightarrow \mathbb{R}^d$  for  $d > 0$  of a mostly continuous physical problem is approximated by a sequence of discrete functions  $U^m \in \mathcal{V}^m$ .

For every  $\mathcal{V}^m$  a specific base  $\mathcal{B}^m = \{\phi_j^m\}_{j \in I^m}$ , for some index set  $I^m$  is used in the numerical method. I. e.  $U^m = \sum_{j \in I^m} \tilde{U}_j^m \phi_j^m$  for a set of coefficients  $\{\tilde{U}_j^m\}_{j \in I^m}$ . Typically the support of a base function  $\phi^m \subset \mathcal{B}^m$  is localized in space, where

$$\text{supp}(\phi) := \{x \in \Omega | \phi(x) \neq 0\}.$$

Here localized does not necessarily mean in each, but at least in some direction, as we will see in what follows. Every function  $\phi^m \in \mathcal{B}^m$  can be uniquely represented on the larger space  $\mathcal{B}^{m+1}$

$$\phi^m(x) = \sum_{\phi^{m+1} \in \mathcal{C}(\phi^m)} \omega(\phi^{m+1}, \phi^m) \phi^{m+1}(x)$$

where the set  $\mathcal{C}(\phi^m)$  of child base functions consists of the  $\phi^{m+1} \in \mathcal{B}^{m+1}$  those supports intersect the support of  $\phi^m$ . Let us emphasize that this definition differs from the frequently used one, where the child set, a subset of ours, contains the higher level base functions with non vanishing weights  $\omega(\phi^{m+1}, \phi^m)$ . Vice versa a parent set is defined by  $\mathcal{P}(\phi^{m+1}) := \{\phi^m \in \mathcal{B}^m | \phi^{m+1} \in \mathcal{C}(\phi^m)\}$ . Let us underline that – in contrast to the case of spatial meshes – there are in general multiple parents.

Frequently the larger space  $\mathcal{V}^{m+1}$  can be decomposed as a direct sum

$$\mathcal{V}^{m+1} = \mathcal{V}^m \oplus \mathcal{W}^{m+1}$$

where the offset space  $\mathcal{W}^{m+1}$  is spanned by a natural base  $\mathcal{B}_+^{m+1}$ . This base consists of hierarchical offset functions  $\psi_i^{m+1}$  for  $i$  in some index set  $I_+^{m+1}$ , which discretize the additional details on the finer approximating space. The union of bases  $\mathcal{B}^0 \cup \mathcal{B}_+^1 \cup \mathcal{B}_+^2 \cup \dots \mathcal{B}_+^{m_{\max}}$  is called a hierarchical base. Let us assume the discrete solution  $U$  to be decomposed in the hierarchical base

$$U(x) = \sum_{i \in I^0} U_i^0 \phi_i^0 + \sum_{1 \leq m \leq m_{\max}} \sum_{i \in I_+^m} U_i^m \psi_i^m(x),$$

where  $U_i^0 = \tilde{U}_i^0$ . The recursive conversion from the standard base to the hierarchical base is called hierarchical analysis, whereas its inverse is denoted hierarchical synthesis.

Finally we assume child and parent base function sets  $\mathcal{C}(\psi)$ , respectively  $\mathcal{P}(\psi)$ , to be defined by analogy.

Now we may ask for a relation between the decompositions in function space and in physical space. Thereby the natural question in post processing is the question for an appropriate decomposition of physical space to localize the computed simulation solution. Artifacts would arise if we try to restrict ourselves to one type of spatial decomposition and project, e. g. subsample, any discrete function onto one, in general totally different, accompanying function space.

The building blocks of the function space with respect to physical space are the in general overlapping supports  $\text{supp}(\phi_j^m)$  of the base functions  $\phi_j^m \in \mathcal{B}^m$ . In most cases, except for gridless discretizations, these supporting domains naturally decompose into cells (elements) of a specific class of hierarchical domain decomposition. The same holds for the hierarchical base functions. Then a canonical strategy in hierarchical data analysis and post processing is to work on this hierarchy:

- visit its grid cells,
- reconstruct the local solution function, and
- apply some local rendering,

such as local isosurface extraction or color shading on slices in 3D. The reconstruction of the local solution function is thereby the central task and should be implemented efficiently. An evaluation of function values on typical discrete numerical function spaces in world coordinates is in general computationally expensive. Here the concept of local coordinates plays an important role. Every element is accompanied by some local coordinate system. The local base functions can easily be evaluated in these coordinates. I. e. a function  $f$  on a specific element  $E$  should be accessed by by a call like  $f(E, c)$  where  $c$  is the local coordinate vector. Fig. 2 depicts several coordinate systems for different types of elements.

Before we explain general hierarchical and adaptive visualization by an appropriate procedural subsampling, let us first inspect – with this aim in mind – typical data bases. We will thereby clarify the concrete post processing tasks.

## 2.1 Finite Volume Methods

The simplest function space on a grid hierarchy, is the space of piecewise constant functions on the cells of each hierarchical level. They especially arise in Finite Volume methods in CFD. On every  $\mathcal{M}^l$  from an arbitrary hierarchy of nested grids the functions  $\phi_{E^l} = \chi_{E^l}$ , where  $\chi_{E^l}$  is the characteristic function on an element  $E^l \in \mathcal{M}^l$ , define the canonical base  $\mathcal{B}^l$  of the corresponding function space. Obviously there is a unique parent relation in the nested function spaces  $\mathcal{V}^l = \text{span } \mathcal{B}^l$  coming along with the single weight  $\omega(\phi^{m+1}, \phi^m) = 1$ .

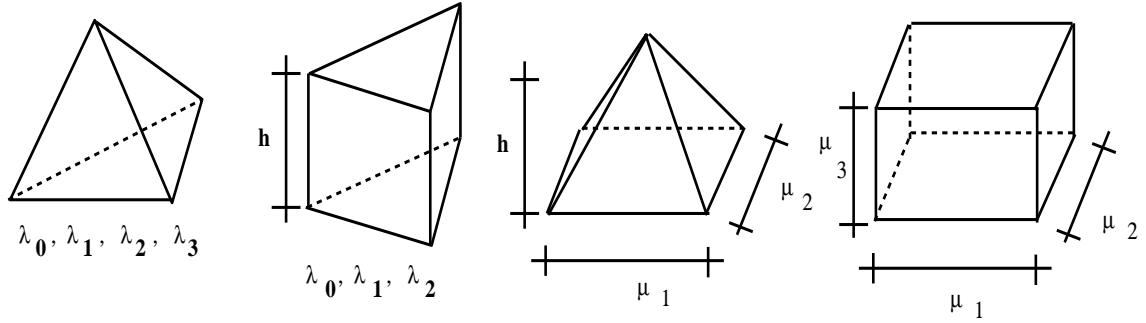


Figure 2: Some possible local coordinate systems of a tetrahedron (barycentric coordinates  $\{\lambda_i\}_{i=0,\dots,3}$ ), a prism (barycentric coordinates  $\{\lambda_i\}_{i=0,\dots,2}$  on the base triangle and height  $h$ ), a pyramid (bilinear coordinates  $\mu_1, \mu_2$  on the base and height  $h$ ) and a cube (trilinear coordinates  $\mu_1, \mu_2, \mu_3$ )

## 2.2 Multilinear Finite Elements

Now we ask for continuous discrete functions on grids, those elements are the above introduced tensor products of simplices. Thereby we are above all lead to the multilinear Finite Element space  $\mathcal{V}$ . I. e. on each element  $E = \bigotimes_{i=1}^k \sigma_i$  the local function space  $\mathcal{V}_E$  is spanned by functions  $U = \prod_i^m U^i$  where  $U^i$  is linear on  $\sigma_i$  [5]. The global discrete space  $\mathcal{V}$  on a conforming grid  $\mathcal{M}$  is then defined by the additional assumption of continuity. An arbitrary function  $U \in \mathcal{V}$  is uniquely given by its values on the set of element vertices  $\mathcal{N}(E)$ , called nodes. Therefore to each node  $x \in \mathcal{N}(E)$  corresponds a base function  $\phi_x \in \mathcal{V}$  which is 1 at  $x$  and 0 on all other nodes. As its support we obtain  $\{E \in \mathcal{M} \mid x \in \mathcal{N}(E)\}$ . Every element  $E$  is the overlap of all base functions  $\phi_x$  corresponding to its nodes  $x$ .

Every node  $x^{l+1} \in \mathcal{N}(E^{l+1})$  for  $E^{l+1} \in \mathcal{M}^{l+1}$  can be expressed as a weighted sum over its thereby defined parent nodes  $\mathcal{P}(x^{l+1}) \subset \mathcal{N}(E^l)$  for  $E^l = \mathcal{P}(E^{l+1})$

$$x^{l+1} = \sum_{x^l \in \mathcal{P}(x^{l+1})} \omega(x^{l+1}, x^l) x^l,$$

with  $\omega(x^{l+1}, x^l) \neq 0$ . Vice versa the set of child nodes is given by  $\mathcal{C}(x^l) := \{x^{l+1} \in \mathcal{N}(\mathcal{M}^{l+1}) \mid x^l \in \mathcal{P}(x^{l+1})\}$ . Finally we obtain the straightforward prolongation operation

$$U(x^{l+1}) = \sum_{x^l \in \mathcal{P}(x^{l+1})} \omega(x^{l+1}, x^l) U(x^l)$$

for  $U \in \mathcal{V}^l$ , and a base function is the following linear combination of its child base functions:

$$\phi_{x^l} = \sum_{x^{l+1} \in \mathcal{C}(x^l)} \omega(x^{l+1}, x^l) \phi_{x^{l+1}}.$$

This indicates that the corresponding hierarchical offset base consists of the base functions, those nodes appear for the first time on the current grid level:

$$\mathcal{B}_+^{l+1} = \{\phi_{x^{l+1}} \mid x^{l+1} \in \mathcal{N}(\mathcal{M}^{l+1}) \setminus \mathcal{N}(\mathcal{M}^l)\}$$

When we recursively traverse the grid hierarchy, we recover the local function on the next finer level by requesting all base function coefficients corresponding to the nodes of the child elements.

## 2.3 Higher order polynomial Finite Elements

The above local space can be enlarged by increasing the polynomial degree of the functions on the initial simplices. We will restrict ourselves here to the case of Lagrangian Finite Elements, where additional nodes  $x$ , which are not necessarily vertices of an element, correspond to additional base functions  $\phi_x$  [5]. Again a discrete

function  $U$  in the corresponding Finite Element space  $\mathcal{V}$  is uniquely defined by its values on the nodes and for a base function  $\phi_x \in \mathcal{V}$  we assume that it equals 1 at  $x$  and vanishes on all other nodes. Therefore its support again consists of the union of those elements which contain the node. A prolongation operation can be derived similar to the multilinear case above. Therefore we analogously define sets of parent and child base functions. There are different ways to construct a hierarchical offset base. As in the multilinear case they are typically a subset of the standard base. The considered construction strongly depends on the numerical method, therefore we do not go into the details here.

A highly efficient hierarchical finite element space is the one which consists of multilinear functions in areas, where error estimates indicate certain singularities in the approximated function, and polynomials of very high order, where the solution is supposed to be sufficiently smooth. This approach is called  $hp$ -Finite Element method (for the gridsize  $h$  and the polynomial degree  $p$ ) [37, 41].

## 2.4 Sparse Grids

An important and recently developed class of numerical methods is based on so called Sparse Grids [3, 11, 47]. The set of nodes of a Sparse Grid is a subset of the full grid nodes (cf. Fig. 3). A regular sparse grid in three dimensions consists of  $O(N \log(N)^2)$  nodes in contrast to the  $N^3$  nodes which are necessary for the corresponding full grid. Nevertheless the interpolation error on a Sparse Grid is only slightly increased to an order of  $O(h^2 \log(h)^2)$  for smooth functions where  $h = \frac{1}{N}$ . This makes Sparse Grid methods prominent approximation methods also in numerical simulation.

For simplicity let us assume  $\Omega = [0, 1]^3$  and define at first a one dimensional hat function  $\hat{\phi} : \mathbb{R} \rightarrow \mathbb{R}$  which is piecewise linear on intervals  $[z, z+1]$  for integer numbers  $z$ , 1 at 0 and vanishing on all other integers. Now for a multiindex  $\alpha \in \mathbb{N}_0^3$  consider a regular grid  $\mathcal{M}^\alpha$  which is of different fineness  $h^\alpha = (2^{-\alpha_1 n}, 2^{-\alpha_2 n}, 2^{-\alpha_3 n})$  in the three principle directions. To each node  $x^\alpha \in \mathcal{N}(\mathcal{M}^\alpha)$  corresponds a base function

$$\phi_{x^\alpha}(y) := \prod_{i=1}^3 \hat{\phi}\left(\frac{y_i - x_i^\alpha}{h^{\alpha_i}}\right)$$

and these base functions are gathered in sets  $\mathcal{B}^\alpha := \{\phi_{x^\alpha} \mid x^\alpha \in \mathcal{N}(\mathcal{M}^\alpha)\}$  for every  $\alpha$ . With these notation at hand we are now able to define the  $m$ 'th Sparse Grid base and function space

$$\begin{aligned} \mathcal{B}^m &:= \{\phi_{x^\alpha} \mid x^\alpha \in \mathcal{N}(\mathcal{M}^\alpha) \wedge \alpha_1 + \alpha_2 + \alpha_3 = m\}, \\ \mathcal{V}^m &:= \text{span } \mathcal{B}^m \end{aligned}$$

and  $\mathcal{N}^m := \bigcup_{\sum \alpha_i = m} \mathcal{N}(\mathcal{M}^\alpha)$  is the accompanying set of nodes (cf. Fig. 3). Corresponding to this tensor product structure, a child relationship is defined.  $\phi_{\tilde{x}^\alpha}$  is a child of  $\phi_{x^\alpha}$  iff  $\tilde{\alpha} = \alpha + e_j$  for  $j \in \{1, 2, 3\}$  and  $\tilde{x}^{\tilde{\alpha}} \in \mathcal{N}(\mathcal{M}^{\tilde{\alpha}}) \cap \text{supp}(\phi_{x^\alpha})$ . According to the above outline the parent relation is defined by duality. The construction of an hierarchical offset base is straightforward. We simply follow the tensor product structure and start with the linear hierarchical base in one dimension. Let us emphasize that a base function support is no longer local in space. If we refine the space  $\mathcal{V}^m$  adding hierarchical offset base functions, we obtain  $\mathcal{V}^{m+1}$ . There are hierarchical offset base functions which are very thin in some direction and fat in the other directions as well as others with equilibrated support in all principle directions.

Once the user has calculated numerical data on a sparse grid he wants to explore the results visually in a post process. The very first attempt to do so consists of an in advance mapping of sparse grid data onto the corresponding standard  $N^n$  grid. But this is definitely ruled out if  $N$  is sufficiently large, typically  $2^{10}$  and larger. We will overcome this shortcoming by the procedural access to the data.

## 2.5 Wavelets and Pre-Wavelets

Wavelet based methods are effective instruments in numerical methods [10], data compression [46] and also in multiresolutional visualization [13, 28, 9]. Thereby the spaces  $\mathcal{V}^m$  from a sequence of nested function spaces  $\mathcal{V}^0 \subset \mathcal{V}^1 \subset \dots \mathcal{V}^{m_{\max}}$  with bases  $\mathcal{B}^m$  are expressed as a direct orthogonal sums

$$\mathcal{V}^m = \mathcal{V}^{m-1} \oplus \mathcal{W}^m, \quad \mathcal{V}^{m-1} \perp \mathcal{W}^m.$$

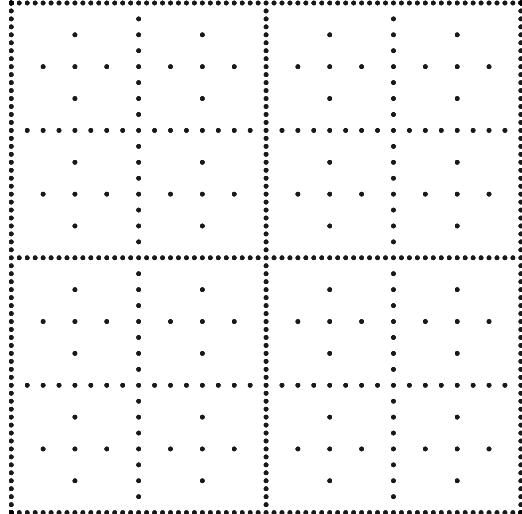


Figure 3: The nodes of a 6 level sparse grid in 2D are dotted.

An orthogonal base  $\mathcal{B}_*^m$  for  $\mathcal{W}^m$  is called a wavelet base consisting of wavelets. Wavelets are therefore special types of hierarchical base functions. If the latter orthogonality is skipped we call the base functions pre-wavelets. Furthermore wavelet analysis is the recursive implementation of the base transformation from the  $\mathcal{B}^{m_{\max}}$  to  $\mathcal{B}^0 \cup \mathcal{B}_*^1 \cup \mathcal{B}_*^2 \cup \dots \cup \mathcal{B}_*^{m_{\max}}$ , and wavelet synthesis is its again recursive inverse. An essential task is to find useful bases  $\mathcal{B}_*^m$ . Typically they are defined on tensor product grids as tensor products of corresponding one dimensional wavelets or pre-wavelets. In the context of biorthogonality similar ideas have been carried out on simplicial grids as well [4, 8, 35].

In the post processing step we want to analyze a discrete function  $U \in \mathcal{V}^{m_{\max}}$  by multiresolutional visualization tools. As for the other data types the final rendering mostly refers to the data representation in the original base, whereas complete data can best be stored in compressed wavelet form. For that reason the wavelet synthesis has to be carried out during the visualization method and wavelet coefficients themselves may deal as the foundation of an accompanying adaptive approach. Beyond that the treatment is analogous to the case of a standard function representation.

## 2.6 Gridless Discretizations, Particle Methods

The bases in the discrete function spaces, discussed so far, are always related to some grid. This relation is skipped in case of Particle Methods or Gridless Discretizations. Here a set  $\Pi$  of density measures or functions  $\pi_i$  with  $i \in I^\pi$  is given, where  $I^\pi$  is some index set. Let  $x_i^\pi$  denote the center of mass of  $\pi_i$  and  $\mathcal{N}^\pi$  the set of these nodes. In general the functions  $\pi_i$  have local support or they are rapidly decreasing in the distance from the node  $x_i^\pi$ . The space  $\mathcal{V}^\pi := \text{span } \Pi$  corresponds to the set  $\Pi$  of density functions.

For true particles these functions  $\pi_i$  can be interpreted as Dirac measures at the positions  $x_i^\pi$ , representing physical quantities. In case of Smooth Particle Hydrodynamics, e. g. in astrophysics, each  $\pi_i$  is a rotationally symmetric function with local support, representing a fluid density, typically not a particle. Gridless Discretization methods ask for Galerkin type solutions in  $\mathcal{V}^\pi$ , where  $\Pi$  is assumed to be a Partition of Unity.

For all these methods a global density

$$U^\pi = \sum_{i \in I^\pi} U_i^\pi \pi_i,$$

is computed as the numerical solution in the space  $\mathcal{V}^\pi$ . At first there is no hierarchical structure accompanying the data. But especially on a larger spatial scale it makes sense to project the functions  $U^\pi \in \mathcal{V}^\pi$  onto a sequence of hierarchically nested spaces  $\mathcal{V}^0 \subset \dots \subset \mathcal{V}^{m_{\max}}$ . If  $\mathcal{V}^m$  is spanned by a base  $\mathcal{B}^m = \{\phi_i^m\}_{i \in I^m}$  which

has the Partition of Unity property then

$$\begin{aligned} U &:= \sum_{i \in I^m} \tilde{U}_i \phi_i^m \quad \text{with} \\ \tilde{U}_i &:= \int_{\Omega} U^\pi \phi_i^m \end{aligned}$$

defines an appropriate projection. In case of Dirac densities  $\tilde{U}_i = \sum_{j \in I^\pi} \phi_i^m(x_j^\pi) U_j^\pi$  holds.

In general, if base functions have supports that are local in space, we only have to sum over a bounded number of contributions. We can analogously apply a hierarchical base, where the corresponding sum can easily be computed in a recursion.

Now instead of  $U^\pi$  the projection  $U$  can then be taken into account for hierarchical post processing purposes. In addition on the finest scale still the original data is available. Here we recognize a strong analogy to the handling of *hp*-Finite Element data in a multiresolutional analysis, where also beyond the finest grid scale we will inspect the true function data.

### 3 Visualization by hierarchical spatial subdivision

So far we have discussed different discrete function spaces as they appear in various efficient numerical methods and their relations to a hierarchical subdivision in physical space. This spatial subdivision is the natural approach for a set of important visualization methods. Prominent examples are the isosurface rendering and the color shading or vector plotting in 2D, respectively on slices in 3D. These methods can be implemented as a recursive traversal of the hierarchy of grid elements with some local rendering technique invoked on the finest cells. A very first, preliminary version of such a recursive algorithm *Inspect()* applied to any macro element is sketched in the following pseudo code:

```
Inspect(E) {
    if ElementOfInterest(E) {
        if C(E)
            for all  $\tilde{E} \in C(E)$ 
                Inspect( $\tilde{E}$ );
        else {
            Project( $U, \mathcal{B}^{m_{\max}}, E$ );
            Extract( $E$ );
        }
    }
}
```

where *ElementOfInterest(E)* is the boolean valued criterion which indicates the presence of features such as local isosurfaces (cf. [45]) and *Project( $U, \mathcal{B}^{m_{\max}}, E$ )* is the evaluation of the local function  $U \in \mathcal{V}^{m_{\max}}$  on a leaf element  $E$ , i. e. the summation over all contributions from base functions  $\phi \in \mathcal{B}^{m_{\max}}$ . Finally the procedure *Extract( $E$ )* performs the actual local rendering on the element  $E$ . Up to now the only benefit from the hierarchical structure is that we can stop on coarse elements on which we already know on the coarser grid level that there are definitely no graphic primitives to be drawn.

Beyond that, the key issue in a multiresolutional visualization strategy is to stop and extract information already on coarse elements and a corresponding locally coarser function space, if an induced error expression is below a certain threshold. If the true data is already sufficiently approximated on a coarse cell, we can skip the expensive search for detailed features to be visualized on the child level. In most cases the true value of the considered error expression is not very handsome for computation. Therefore frequently some computable bound or error indicator is taken into account. If  $\eta(E)$  denotes such an indicator for an element  $E$  and  $\epsilon$  is an user prescribed error tolerance,  $\mathcal{S}(E) := (\eta(E) \leq \epsilon)$  is a possible stopping criterion in the algorithm.

If we apply this stopping criterion in the above algorithm, graphical primitives are extracted on locally different levels of detail in physical space (cf. Fig. 4). I. e. at transition faces between leaf elements on different grid levels of a recursive traversal, different sets of base functions and thereby different approximations of the true function  $U$  are taken into account. Thereby it is obvious that this rudimental strategy comes along with the

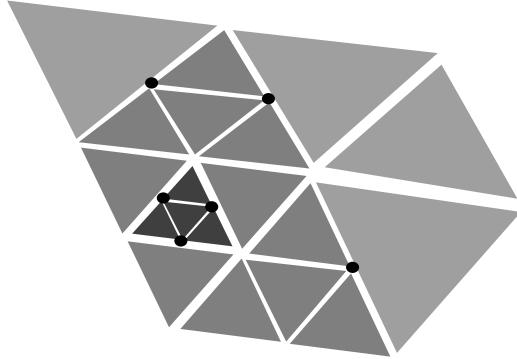


Figure 4: An adaptive traversal of a 2D grid leads to non conforming grids, respectively hanging nodes.

drawback of cracks in isolines or jumps in the color intensity at edges in 2D or on slices in 3D, respectively cracks in isosurfaces in 3D. At first one might ask for some adjustment procedure, which explicitly refers to adjacency information among elements on the same or on different grid levels (cf. [39]). But then the adaptive visualization method can no longer be coded as a strict, easy to implement and fast depth first hierarchical tree traversal.

A closer look onto the above shortcoming points out its intrinsic reason: The support of the base functions overlap multiple elements on corresponding grid levels. Therefore we have to ensure that base functions considered on a specific element have also to be taken into account on any other element which intersects the base function support. In what follows let us assume that data is always represented in the hierarchical base, otherwise we have to convert it. Furthermore the error measurement should be related to these hierarchical base functions instead of the elements. It should indicate whether the detailed information of a base function could be neglected or not. Then the adaptive projection of the true data, which is definitely visualized, will turn out to be continuous and so its graphical representation will be visually acceptable. We are left to realize this concept in a concrete algorithms.

First of all, we replace the stopping criterion on elements by some boolean valued *projection criterion*  $\mathcal{S}(\psi^m)$  for every hierarchical offset function  $\psi^m \in \mathcal{B}_+^m$  with  $m \in I_+$ . If  $\eta(\psi^m)$  is an error indicator corresponding to  $\psi^m$  and  $\epsilon$  is again the user prescribed threshold then we define

$$\mathcal{S}(\psi^m) := (\eta(\psi^m) \leq \epsilon).$$

Next, for each type of numerical data coded in a nested sequence of function spaces  $\{\mathcal{V}^m\}_{m \leq m_{\max}}$  we choose a corresponding sequence of nested grids  $\{\mathcal{M}^l\}_{l \leq l_{\max}}$  and relate to each  $\mathcal{M}^l$  the data approximation in  $\mathcal{V}^{m(l)}$ . Here  $m(\cdot)$  is a uniquely defined mapping, but it is not necessarily injective. For instance trilinear functions on a nested hexahedral grid can be explored on an octree, deviding a cell into eight child cells ( $m(l) = l$ ) or by cyclic bisection in the coordinate directions ( $m(l) = l \text{ div } 3$ ). A visualization algorithm now recursively traverses elements in the sequence of grids and recovers on each element an approximation of the numerical solution  $U$ . With an error indicator  $\mathcal{S}(\cdot)$  at hand this approximation can be chosen as a continuous, adaptive projection  $P_{\mathcal{S}}^l$ . If we denote by  $\mathcal{B}_+^m(E)$ ,  $\mathcal{B}^0(E)$  the subsets of  $\mathcal{B}_+^m$ , respectively  $\mathcal{B}^0$ , those elements have supports which intersect the element  $E \in \mathcal{M}^l$ , we can recursively define

$$P_{\mathcal{S}}^l U|_E(x) := P_{\mathcal{S}}^{l-1} U|_{\mathcal{P}(E)} + \sum_{\substack{\psi_i^m \in \mathcal{B}_+^{m(l)}(E) \\ \neg \mathcal{S}(\psi_i^m)}} U_i^m \psi_i^m(x),$$

where  $P_{\mathcal{S}}^0 U|_E(x) = \sum_{\phi_i^0 \in \mathcal{B}^0(E)} U_i^0 \phi_i^0(x)$ . If  $\mathcal{S}(\psi)$  is true for all  $\psi \in \mathcal{B}_+^m(E)$  with  $m > m(l)$  then

$$P_{\mathcal{S}}^k U|_E = P_{\mathcal{S}}^l U|_E$$

for all  $k$  with  $m(k) \geq m(l)$ , that is the projection keeps unchanged if we recursively process child elements on finer grid levels. This implies a natural *stopping criterion* on elements  $E \in \mathcal{M}^l$

$$\mathcal{S}(E) := \bigwedge_{\psi \in \mathcal{B}_+^m(E), m > m(l)} \mathcal{S}(\psi).$$

Although the adaptive projection is continuous by definition in case of isosurfaces on specific 3D grid types, we have to handle carefully the restriction of  $P_S^l U$  at transition faces between different levels of resolution, on which bilinear discrete functions are involved (cf. Sect. 6). Checking for the element stopping criterion implies testing the projection criteria for all fine scale, hierarchical base functions those supports of which intersect the element. This would require an enormous look ahead, which is computationally expensive and not very handsome. Therefore we require a natural saturation condition for the projection criterion:

**(Saturation Condition)** *If the projection criterion  $\mathcal{S}(\psi)$  is true then  $\mathcal{S}(\hat{\psi})$  is also true for all  $\hat{\psi} \in \mathcal{C}(\psi)$ .*

We will call a projection criterion, respectively an error indicator admissible, if this saturation condition is fulfilled. Based on it the stopping criterion simplifies to

$$\mathcal{S}(E) := \bigwedge_{\psi \in \mathcal{B}_+^{m+1}(E)} \mathcal{S}(\psi),$$

because  $\mathcal{S}(\psi)$  for  $\psi \in \mathcal{B}_+^{m(l)}$  implies  $\mathcal{S}(\psi)$  for  $\psi \in \mathcal{B}_+^m$  and  $m > m(l)$ . If the saturation condition is not fulfilled for a specific type of error indicator, then we can adjust the indicator values in a preprocessing step. Thereby we in general only slightly blow up the error indicators on the coarser hierarchical base functions (cf. Sect. 6). Such a saturation condition is very often implicitly assumed in multiresolutional visualization. Here we state it explicitly. This especially prevents us from overlooking details on fine grid levels.

Now we are able to formulate an abstract multiresolutional algorithm. We will later on comment on implementational aspects for different types of numerical data. The algorithm recursively computes an adaptive data projection  $U^*$ , checks for elements of interest as above and finally invokes the local rendering on an appropriate grid level:

```
Inspect( $U^*, E$ ) {
     $U^* := U^* + \text{AdaptiveOffset}(U, \mathcal{B}_+(E), E);$ 
    if ElementOfInterest( $E$ ) {
        if  $\mathcal{C}(E) \neq \emptyset \wedge \neg \mathcal{S}(E)$ 
            for all  $\tilde{E} \in \mathcal{C}(E)$ 
                Inspect( $U^*|_{\tilde{E}}, \tilde{E}$ );
            else Extract( $E$ );
    }
}
```

Here *AdaptiveOffset()* evaluates the contributions of all offset base functions from  $\mathcal{B}_+(E)$  which do not fulfill the projection criterion. Thereby  $\mathcal{B}_+(E) := \mathcal{B}_+^{m(l)}(E)$  for the corresponding  $m(l)$ .

The *ElementOfInterest()* function typically is based on the data projection  $U^*$  on the current grid level and the error indicator values  $\eta_H(\psi)$  for  $\psi \in \mathcal{B}_+(E)$ . E. g. if the hierarchical offset base is subset of a Partition of Unity which is frequently the case and we recursively define  $\eta_H(\psi_i^m)$  for  $\psi_i^m \in \mathcal{B}_+^m(E)$  by

$$\eta_H(\psi) := |U_i^m| + \max_{\hat{\psi} \in \mathcal{C}(\psi)} \eta(\hat{\psi}),$$

where the second term is skipped on the finest level of the hierarchy, the true values of the numerical solution  $U$  are estimated by

$$\min_E U^* - \max_{\psi \in \mathcal{B}_+(E)} \eta_H(\psi) \leq U \leq \max_E U^* + \max_{\psi \in \mathcal{B}_+(E)} \eta_H(\psi).$$

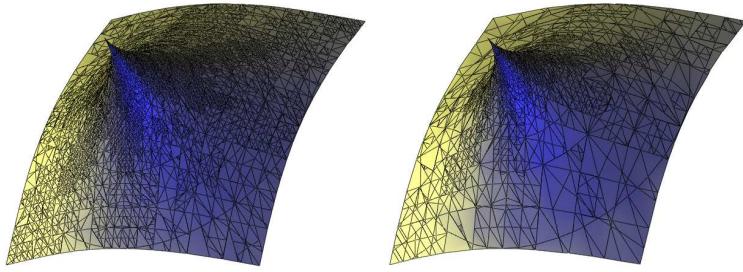


Figure 5: Adaptive isosurface extraction on a test data set for different threshold values; the underlying grid consists of 12 million tetrahedrons.

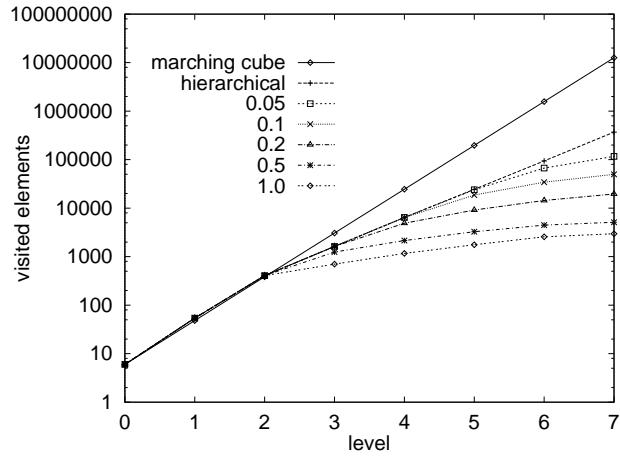


Figure 6: On a logarithmic scale we compare different visualization strategies concerning the overall number of visited cells for increasing grid level. Compare Fig. 5 for the corresponding data set. We expect the purely hierarchical isosurface extraction to distinguish from the marching cubes method in the slope of the curve by a factor of  $\frac{2}{3}$ . This is obviously reflected by the above diagram. Furthermore for successively increased threshold value the method reaches successively earlier a saturated state. The behaviour at the singularity is not visible in the diagram.

We can apply this simple test as a robust checking criterion for isosurfaces on 3D domains. Thereby we avoid the expensive storing of min/max-values as discussed in [45].

Let us finally summarize. Running the adaptive algorithm, the full grid hierarchy is partially traversed (cf. Fig. 5,6). Obviously we can not do better, i. e. resolve the considered physical quantity finer than provided by the actual local depth of the hierarchical data base. Then the adaptive projection in the post processing resamples the computational grid history for decreasing threshold value  $\epsilon$  down to the threshold value  $\hat{\epsilon}$  at which the computation was finally stopped. Fig. 7 depicts this schematically and compares it to a simple cut off on the grid hierarchy at some level. If we use a different projection criterion in the adaptive numerical algorithm and the post processing, the computational grid hierarchy and the portion of it traversed during the visualization algorithm will not match properly. In certain local areas a recomputing would be necessary to overcome this shortcoming.

### 3.1 Finite Volume Methods

Finite Volume data is piecewise constant on elements and discontinuous at element faces. Therefore the projection criterion  $S(\phi_E)$  coincides with the element stopping criterion  $S(E)$ . If we restrict ourselves to the true data and do not ask for an interpolation, continuity of the adaptive projection is ruled out. Nevertheless the

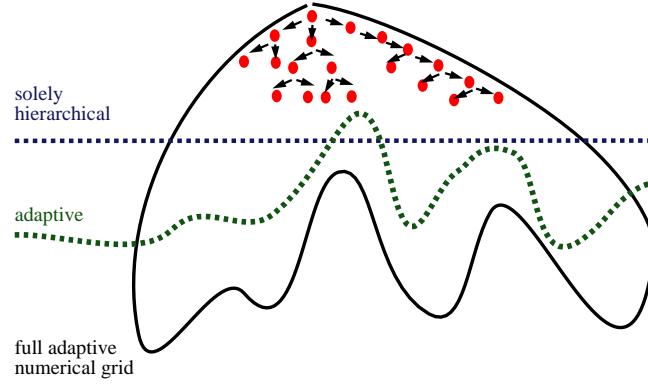


Figure 7: A solely hierarchical traversal of the grid would stop at a certain level of the hierarchy, whereas an adaptive traversal allows a stopping criterion depending on the data.

saturation condition in the post processing makes sense. It guarantees that we do not overlook details on much finer grid levels, which are not detected on the coarse grid data base.

### 3.2 Multilinear Finite Elements

In case of multilinear Finite Elements we have already pointed out a strong relation between base functions and vertices of elements in the grid hierarchy. This induces a straightforward implementation of the visualization method sketched in pseudo code. During the recursive grid traversal we store the possibly projected function values  $P_S U$  on each nodal point  $x^l$ . Data already been calculated on the coarse grid vertices is inherited to the next finer level. On those nodes  $x^l$ , which appear for the first time on the current level  $l + 1$  we check for the corresponding nodal projection criterion  $S(x^l)$ , respectively error indicator  $\eta(x^l)$ , request the function value on  $x^l$  from the data base, or interpolate values from the parent grid level by

$$P_S U(x^l) = \sum_{x^{l-1} \in \mathcal{P}(x^l)} \omega(x^l, x^{l-1}) P_S U(x^{l-1})$$

where the weights  $\omega(x^l, x^{l-1})$  solely depend on the refinement rule and can be retrieved from some simple lookup table. The stopping criterion on elements is evaluated by a one level lookup onto the projection criterion stored on new nodes on the next grid level. Fig. 8 demonstrates the applicability for true simulation data.

### 3.3 Higher order polynomial Finite Elements

Up to now we have solely considered Finite Element data which is for each element in the function space spanned by tensor products of linear functions. Let us now consider post processing, when the local discrete space is enlarged by polynomials of higher degree, as it is especially the case for  $hp$ -Finite Element methods. The aim is to recover the better approximation quality known from the numerical method also in the visual appearance. Here we discuss a generalization of the presented adaptive approach and continue the spatial subdivision beyond the finest grid level. Therefore error information is measured on leaf elements of the actual grid hierarchy and if necessary additional ‘‘virtual’’ grid levels are introduced. On the corresponding ‘‘virtual’’ elements  $E_{hp}$  we define standard adaptive projections  $P_{S_{hp}}$  corresponding to some projection criterion  $S_{hp}$  on ‘‘virtual’’ nodes  $x_{hp}$ . The corresponding function spaces are again spanned by multilinear functions but now on the procedurally refined grid. We can interpret this strategy as an  $h$ -subsampling of the actual polynomial data. On the new grid nodes error indicators are computed, not stored. If such an preliminary error indicator  $\tilde{\eta}_{hp}$  is uniquely defined by the true function value on the node and on its parent nodes, the computational costs can be kept low and independent of the currently inspected ‘‘virtual’’ element. Finally we modify this indicator and obtain by the recursion

$$\eta_{hp}(x_{hp}^{l+1}) := \min\{\tilde{\eta}_{hp}(x_{hp}^{l+1}), \min_{x_{hp}^l \in \mathcal{P}(x_{hp}^{l+1})} \{\eta_{hp}(x_{hp}^l)\}\}.$$

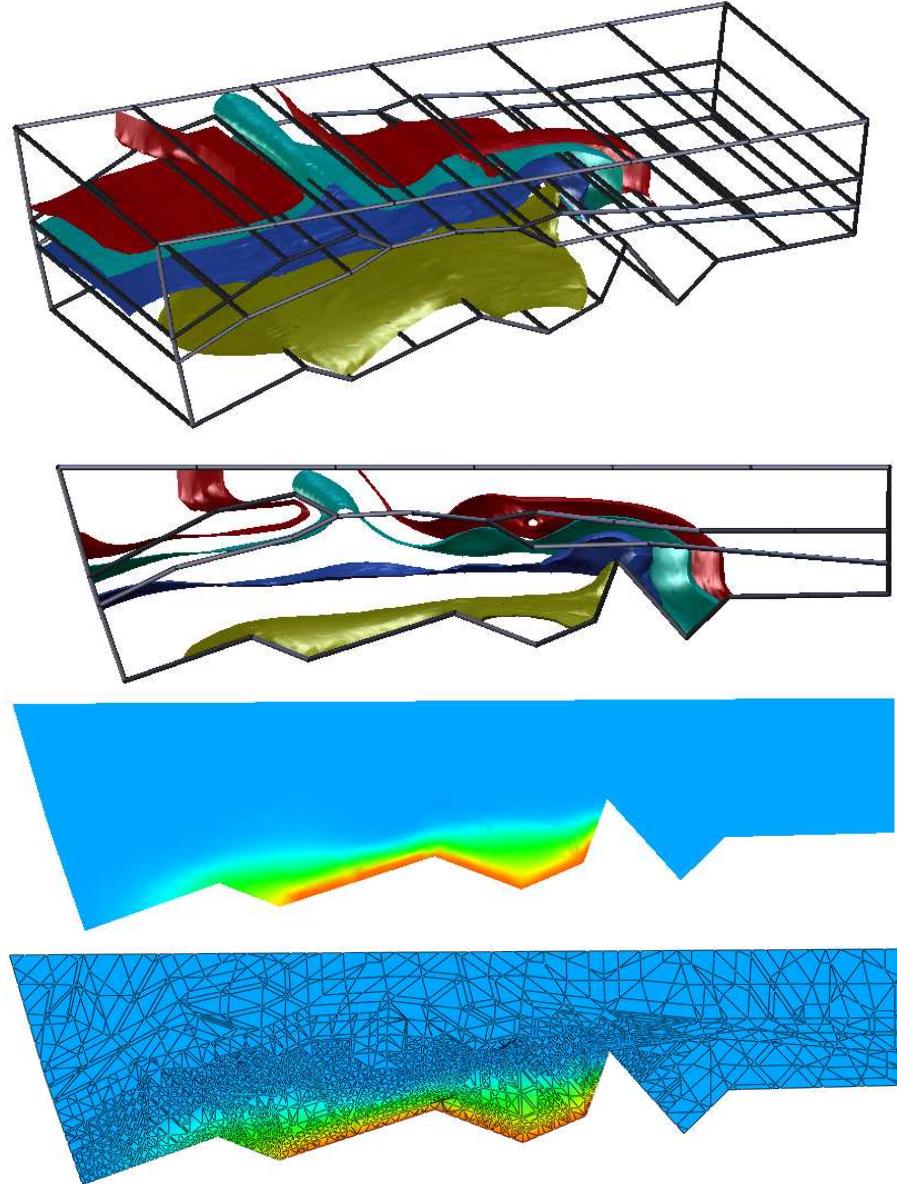


Figure 8: On the top several adaptive isosurfaces are extracted from the simulation data set of a density driven flow. The grid consists of 1.8 million tetrahedrons. In the middle adaptive slicing and color shading is demonstrated and on the bottom in addition the intersections with element faces is outlined in black.

a new indicator, which fulfills the saturation condition by construction. Thereby it is especially ensured that “virtual” nodal values on element faces are always generated by interpolation, whenever the algorithm stops at a coarser level on the corresponding adjacent element. In a certain sense we construct a somewhat saturated error indicator in a top down manner instead of assuming the bottom up implication of the saturation condition. We pay for this simplification by possibly overlooking fine details in the data, stopping on insufficiently refined “virtual” elements. Nevertheless experimental results, which seem to rely on the sufficient smoothness of the considered data function, point out satisfying results. In case of isoline drawing or color shading on slices in 3D we can simplify the algorithm. Instead of procedural subdivision in three dimensions we first compute the intersection polygons and then subdivide them into triangles. Fig. 9 demonstrates the significant improvement in data resolution obtained by this approach.

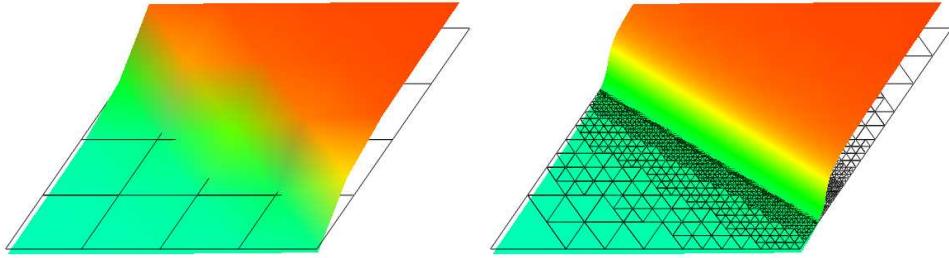


Figure 9: On the left the color shading on a slice, applied to a 3D data set with an only  $h$ -refined grid and on the right the “virtual”  $h$ -subsampling on the same coarse grid.

### 3.4 Sparse Grids

Sparse grid function spaces are subspaces of trilinear Finite Element function spaces of regular hexahedral grids. Therefore the interpolation scheme on a corresponding regular full grid, on which we run the visualization, is obvious. But the sparseness allows grids of very small grid size (cf. Section 2.4). So we have to perform the data interpolation on the majority of grid points during the recursive traversal of the full grid hierarchy in the visualization method. Each value on a full grid node thereby typically depends on  $\log(h)^2$  coefficients of hierarchical sparse grid base functions. If we recalculated them every time, frame rates would drastically be cut down. We recognize that a significant portion of the sum of all base functions contributing to one specific full grid nodal value can be reused on neighbouring grid points. Therefore a caching concept for these linear combinations of sparse grid base functions is a key for an efficient hierarchical subdivision strategy. Each time a nodal value is requested, we search for a suitable cache entry, cut superfluous portions and add necessary additional details. For details on this strategy we refer to [19]. If we combine this with a fixed, small background full grid, on which every nodal value can be stored, we obtain an effective approach to handle interactively even very fine sparse grids. Fig. 10 depicts several significant examples.

### 3.5 Gridless Discretizations, Particle Methods

Large particle sets or gridless discretization have to be interpolated on a nested sequence of function spaces in preparation for an efficient multiresolutional visualization. We choose piecewise trilinear, hierarchical Finite Elements on an adaptive octtree which covers the spatial range of the corresponding density measures.

## 4 Projection criteria and error indicators

The adaptive visualization approach introduced so far is based on some projection criterion  $\mathcal{S}(\psi)$  on hierarchical base functions  $\psi$ . It is still some abstract boolean valued function which is admissible if the saturation condition is fulfilled. Corresponding to different aims of a multiresolutional strategy a variety of concrete projection criteria depending on error indicators, grid and geometry can be defined:

- The almost trivial choice is the *level wise post processing* which is induced by the projection criterion  $\mathcal{S}(\psi^m) = (m > m^*)$  for  $\psi^m \in \mathcal{B}_+^m$  where  $m^*$  is the considered recursion depth.
- The visual impression and a sufficient resolution of numerical data in the visualization process is closely related to curvature, for instance curvature of isosurfaces or isolines on slices. For multilinear Finite Element data the jump of the normalized gradient, denoted by  $[\frac{\nabla U^l}{|\nabla U^l|}]_F$ , locally measures a *discrete curvature*, i. e. the fold in the data function (cf. Fig. 11) at faces. Here the jump operator applied to some function  $W$  is given by  $[W]_F := \lim_{i \rightarrow \infty} |W(x_i^+) - W(x_i^-)|$  for sequences  $\{x_i^-\}$  and  $\{x_i^+\}$  converging to  $x$  from different sides of  $F$ . Therefore we define  $\mathcal{S}(\psi_x)$  corresponding to the error indicator

$$\eta_V(\psi_x) := \max_{\substack{F \in \mathcal{F}(\mathcal{M}^l) \\ \text{supp}(\psi_x) \cap F \neq \emptyset}} \left[ \frac{\nabla U^l(x)}{|\nabla U^l(x)|} \right]_F,$$

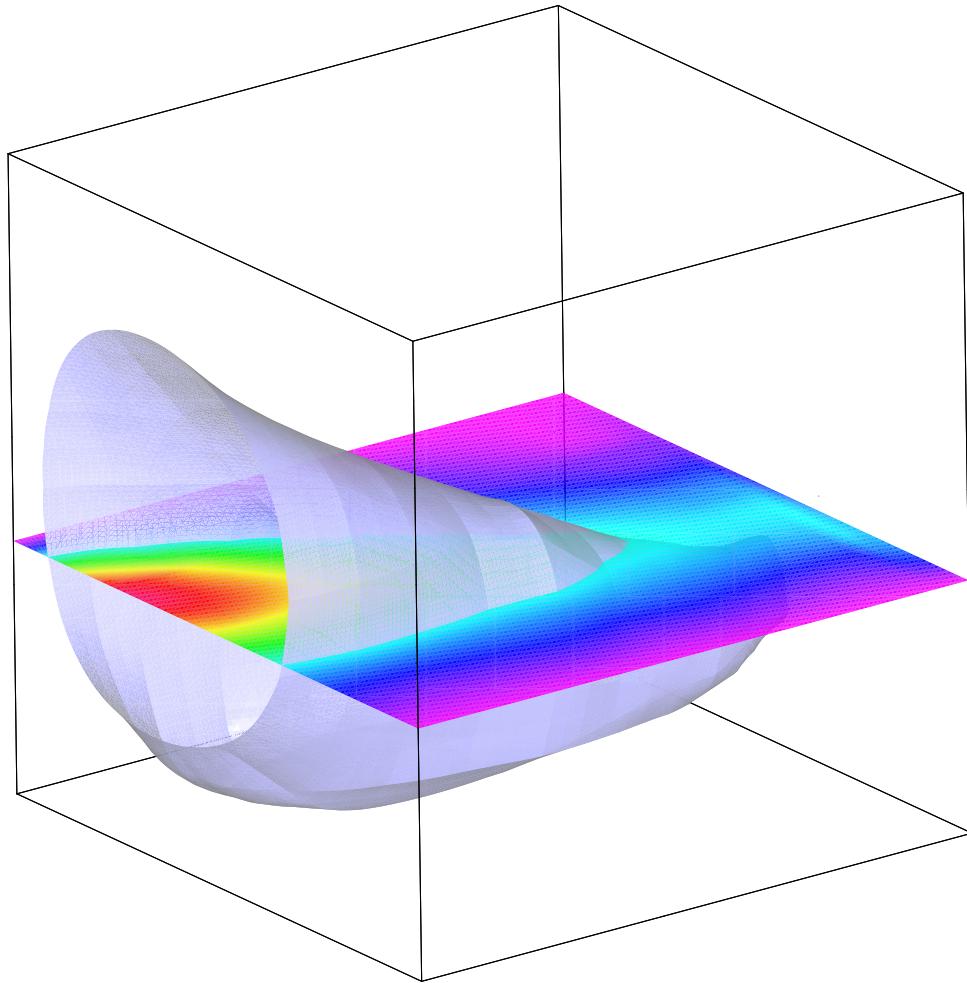


Figure 10: Isosurface rendering and color shading on slices is shown for a 10 level sparse grid data set in 3D.

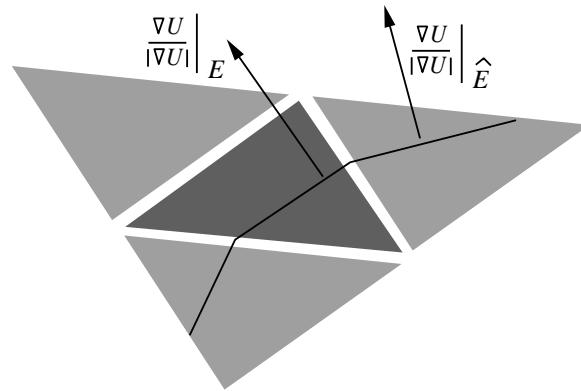


Figure 11: A schematic sketch of the jump of the normalized gradients across an edge in 2D.

where  $U^i$  is the approximation of  $U$  in  $\mathcal{V}^i$  and  $\mathcal{F}(\mathcal{M}^i)$  the set of faces of the corresponding grid. For details we refer to [31].

- Especially in the last decade adaptive numerical methods became popular and proved to be efficient strategies to adequately resolve solution features in simulation computations. They successively compute

an approximate solution in some discrete function space, estimate errors, frequently related to hierarchical offset base functions, and increase the space by those base functions with the largest error contribution. In the post processing we can reuse these *numerical error estimator* values and the deduced projection criteria up to the saturation condition. E. g. for Poisson's problem, computed with piecewise linear elements on simplices,

$$\eta_N(\psi) := h_F \|\nabla U \cdot n_F\|_{L^\infty(F)}$$

serves as an robust error estimator on an element face  $F$  with diameter  $h_F$ . Here  $U$  is the current Finite Element solution,  $n_F$  is the unit normal to  $F$  and  $\psi$  a finer base function with a support intersecting  $F$ .

- Because of the vanishing moments of wavelets or pre-wavelets  $\psi$  their coefficients, which result from an approximation of a numerical solution in a wavelet base, allow an interpretation in terms of “local frequencies”, which often is a desirable feature, e. g. for geographic maps. These *wavelet coefficients* can serve as an appropriate error indicator  $\eta_w(x)$ , if we ensure the saturation condition. Therefore this important class of methods also fits into the presented frame. For a detailed discussion of this topic we refer to the variety of efficient and specialized methods in the literature.
- Instead of one error indicator for a scalar valued function on the domain  $\Omega$  we can also ask for an error indicator for a *vector valued* function. If the same hierarchical base functions are applied for each vector component and  $(\eta_1(\psi), \eta_2(\psi), \dots, \eta_d(\psi))$  is a vector of admissible error indicators then every function  $\gamma : \mathbb{R}^d \rightarrow \mathbb{R}$ , which is monotonely increasing in all its components, applied to this vector is an admissible indicator. Furthermore if a *geometric error control* of some surface  $\mathcal{G}$  is required, the same definition is possible for a parametric representation, where the parametrization plays the role of a scalar or vector valued function over some parameter space. Fig. 12 shows results for some geographical map.

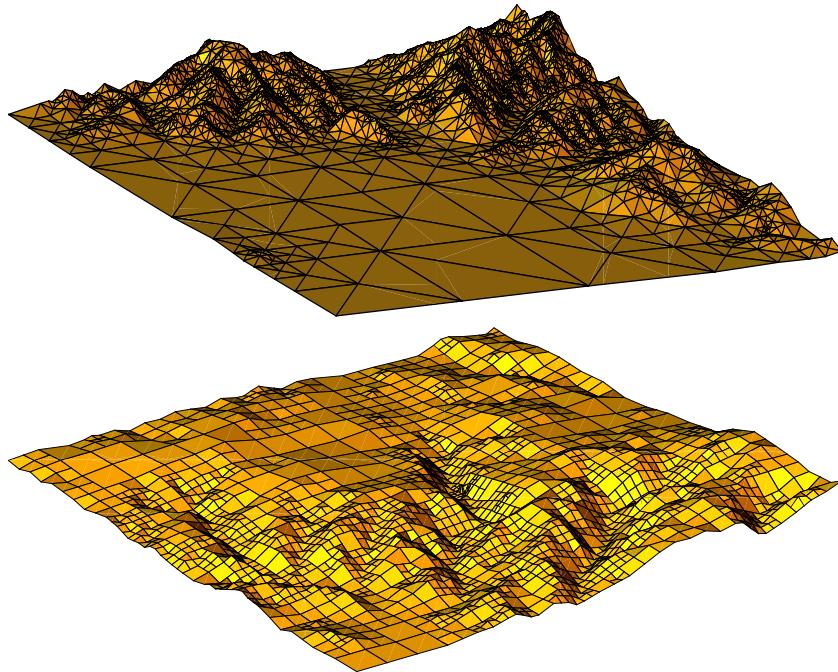


Figure 12: Geometry error control for a geographic map. We compare the data representation on a regular and on a triangular mesh. The elements on which the stopping criterion is fulfilled are outlined for the two grid types.

If several admissible projection criteria  $\mathcal{S}_1, \dots, \mathcal{S}_r$  are supported on a specific data set, they can be *combined in one admissible criterion*:

$$\mathcal{S}_1 \wedge \mathcal{S}_2 \wedge \dots \wedge \mathcal{S}_r$$

Finally let us comment on the handling of the projection criteria in case of multilinear finite elements. Typically function data is stored in arrays, furthermore the hierarchical tree structure representing the grid hierarchy refers to this arrays by indices respectively pointers. Then the error indicator information can be stored in a second array respectively in an enlarged data structure accompanying the original data. For hints on the implementation of the element stopping criteria and the adaptive projection we refer to Section 3.2.

## 5 Timedependent Data

In most physical simulations and for many applications in geometric modeling the discrete function  $U$  or the geometry  $\mathcal{G}$  is time dependent. Typically a sequence of time steps, also called key frames, is given and in between an appropriate interpolation is used. At first we rule out adaptive function spaces in time and assume that data resides in a fixed function space, given by coefficients which correspond to a fixed hierarchical base. We restrict ourselves here to the case of scalar functions. As already mentioned in Paragraph 4 a geometric multiscale analysis works analogously. Let us denote by  $\{U_{t_i}\}_{1 \leq i \leq q}$  the sequence of time steps.  $U(t, x)$  is uniquely defined by an interpolation

$$U_i^m(t) := I(t, U_{i,t_1}^m, \dots, U_{i,t_q}^m)$$

corresponding to the base functions  $\psi_i^m \in \mathcal{B}_+^m$ . The concrete interpolation scheme depends on the application. On the set of time steps  $\{U_{t_i}\}_{1 \leq i \leq q}$  a corresponding set of admissible indicators  $\{\eta_j(\psi_i^m)\}_{1 \leq j \leq m}$  is supposed to be given. We ask for an appropriate indicator  $\eta(t, \dots)$  which again should be admissible. As we already know from Section 4 the expression  $\gamma_t(\eta_1(x), \dots, \eta_q(x))$  is an indicator admissible for any family of component wise increasing functions  $\gamma_t$  with  $t \in [t_1, t_q]$ . If we suppose the interpolation to be defined as a weighted sum

$$I(t, U_{t_1}(x), \dots, U_{t_q}(x)) := \sum_{i=1}^q \mu_{t_i}(t) U_{t_i}(x)$$

with continuous non negative weights  $\mu_{t_i}(t)$  we gain an appropriate induced and admissible indicator

$$\eta(t, \cdot) := \gamma((\mu_{t_i}(t) \eta_{t_i}(\cdot))_{1 \leq i \leq q})$$

where  $\gamma$  is a standard norm in  $\mathbb{R}^q$ , fixed in time (cf. Fig. 13). For the linear interpolation in time the weights coincide with the simple hat functions

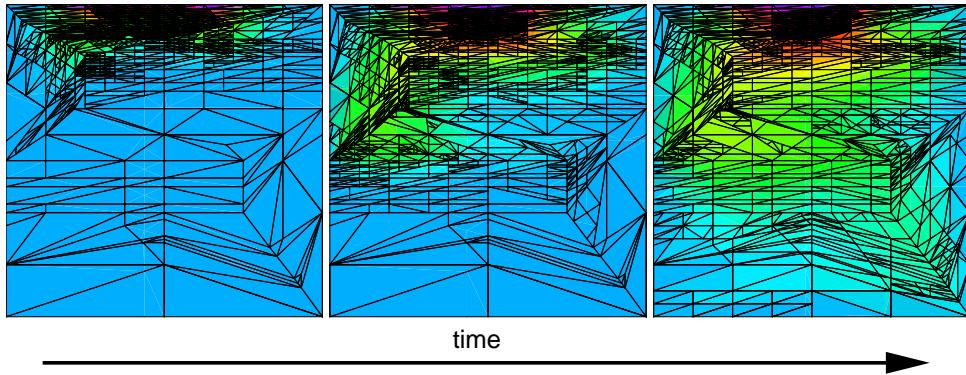


Figure 13: Interpolation in time for adaptive color shading on slices: The pictures show the distribution of the concentration in a two phase flow calculation at different time steps, which do not coincide with timesteps from the computation.

$$\mu_{t_i}(t) := \max\{0, \min\left\{\frac{t - t_{i-1}}{t_i - t_{i-1}}, \frac{t - t_{i+1}}{t_i - t_{i+1}}\right\}\}.$$

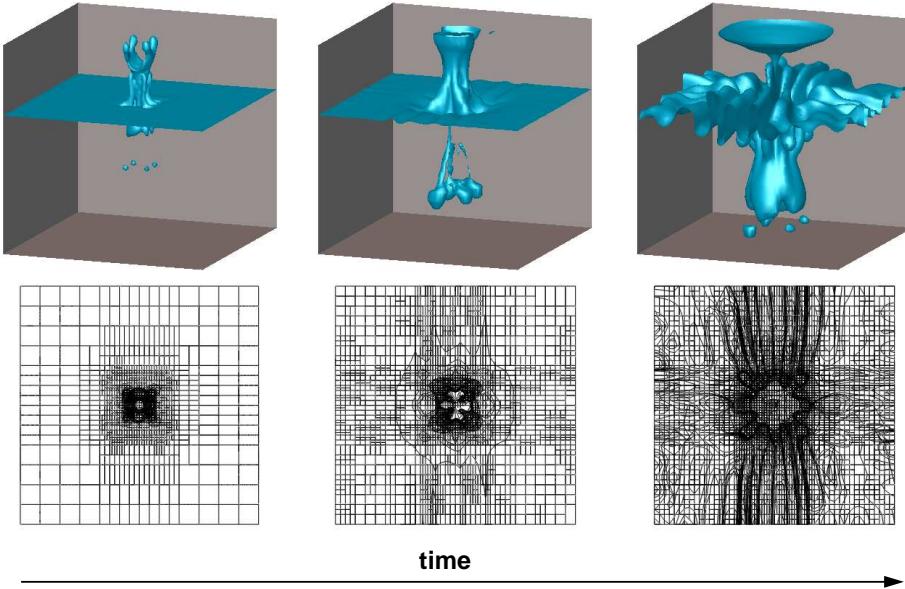


Figure 14: Interpolation in time for adaptive isosurfaces. Above at different times an isosurface of a porous media density is extracted on a hexahedral grid. Below in a projective view from the top the edges of the cells on which the local isosurfaces are extracted show the adaptive approach.

Compare Fig. 13 for an adaptive color shading on slices on a tetrahedral mesh and Fig. 14 for adaptive isosurfaces on hexahedral grids, both extracted from interpolated data. Let us remark that if we take into account the indicator  $\eta_H(\cdot)$  as defined in Section 3 the induced indicator

$$\eta_H(t, \cdot) := \sum_{i=1}^q \mu_{t_i}(t) \eta_{H,t_i}(\cdot)$$

can be used to calculate reliable data bounds for the interpolated function. To check this, we straightforward estimate the difference  $U(t, x) - U^m(t, x)$ , where  $U^m(t, x)$  is the local restriction to the coarser function space  $\mathcal{V}^{m(l)}$  on Element  $E$ .

$$\begin{aligned} & |U(t, x) - U^m(t, x)| \\ &= |I(t, U_{t_1}, \dots, U_{t_m}) - I(t, U_{t_1}^l, \dots, U_{t_q}^l)| \\ &\leq \sum_{i=1}^q \mu_{t_i}(t) |U_{t_i}(x) - U_{t_i}^l(x)| \leq \max_{\psi \in \mathcal{B}_+(E)} \sum_{i=1}^q \mu_{t_i}(t) \eta_{H,t_i}(\psi) \end{aligned}$$

From our point of view this important property points out a significant advantage of the hierarchical intersection test compared to other acceleration algorithms for isosurface extraction. Other techniques are for instance the span space methods [40], the  $k$ -tree method [26] or the extremal graph approach [21]. Without any sophisticated adjustment the expensive preparatory step which comes along with these algorithms has to be invoked on every new interpolation in time. This turns out to a major drawback compared to nearly no extra cost for the hierarchical strategy provided time dependent data is considered.

Let us finally comment on the case of adaptive data in time. Frequently recent numerical methods to solve partial differential equations are based on locally adaptive function spaces defined on adaptive grids. Adaptation is thereby an efficient tool to improve the approximation quality especially for timedeependent applications. Thereby on each interval  $[t_i, t_{i+1}]$  in time a different function space is used to solve the corresponding problem. Let us denote by  $\mathcal{B}_{i,+}$  its hierarchical base. Starting with a solution  $U_i^+$  at time  $t_i$  the numerical algorithm calculates a solution  $U_{i+1}^-$ . Then error indicators are applied to choose a new partially refined or coarsened grid

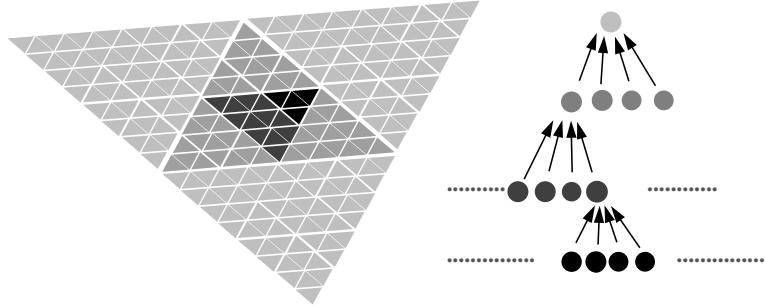


Figure 15: A schematic sketch of the preroll to adjust indicator values and ensure the saturation condition.

and a projection  $U_{i+1}^+$  of  $U_{i+1}^-$  is computed correspondingly. Based on the projected solution the next time step calculation can be performed. If we restrict ourselves to the standard linear interpolation operation  $I(t, \cdot)$  as introduced above the multiscale analysis immediately works on

$$\mu_{t_i}(t)U_i^+ + \mu_{t_i}(t)U_{i+1}^-$$

corresponding to error indicators  $\eta^+(\cdot)$  respectively  $\eta^-(\cdot)$  on the base function  $\psi \in \mathcal{B}_{i,+}$ .

## 6 Further Aspects on the Adaptive Strategy

We have already discussed the importance of the saturation condition a projection criteria, respectively error indicator has to fulfill to ensure continuity of the corresponding adaptive projection. If the saturation condition is not fulfilled for a specific type of projection criterion, then in a preprocessing step we can adjust the criterion. In case of a typical error indicator this in general turns out to be necessary only on coarse grid levels. Whereas on finer grid levels we are already in a saturated state, except at singularities approximated in the data which are still not well resolved. For a detailed numerical background we refer to [1, 2]. If a certain error indicator does not fulfill the above condition, a slight modification leads to a properly saturated indicator, respectively induced projection criterion. A simple update algorithm for an error indicator  $\eta$  and thereby the corresponding projection criterion  $\mathcal{S}_\eta$  is the following level-wise traversal of the hierarchical base functions, starting on the second finest level and ending on the macro grid (cf. Fig. 15).

```

for ( $m = m_{\max} - 1; m \geq 0; m--$ )
  for all  $\psi_i^m \in \mathcal{B}_+^m$  {
     $\eta^* = \max_{\psi \in \mathcal{C}(\psi_i^m)} \eta(\psi);$ 
    if ( $\eta(\psi_i^m) < \eta^*$ )  $\eta(\psi_i^m) = \eta^*$ ;
  }
}

```

Let us emphasize that a depth first traversal of the hierarchy in the adjustment procedure would not be sufficient.

The saturation condition comes along with another desirable and straightforward consequence. Performing the adaptive visualization algorithm we end up with a smooth transition in the fineness of grids leaf cells on which we finally extract graphical primitives. In case of multilinear finite elements we have most one level transitions at faces of elements on which the local rendering takes place. I. e. on each such face vertices of only two, not necessarily successive levels will occur (cf. Fig. 4). We suppose the proof to be instructive concerning the implications of the saturation condition therefore it is given in the following paragraph.

Let us suppose that two elements  $E, \tilde{E}$  meet at a certain face  $F$ . On  $\tilde{E}$  the above algorithm already stops, i. e.  $\mathcal{S}(\tilde{E})$  is true, whereas on  $E$  elements  $E^1 \subset E^2 \subset E$  of two different finer levels are traversed, i. e.  $\mathcal{S}(E), \mathcal{S}(E^2)$  are false (cf. Fig. 16). Furthermore we assume that  $E^1, E^2$  have faces  $F^1$ , respectively  $F^2$  with  $F^1 \subset F^2 \subset F$ . By assumption there exists a node  $x^1 \in \mathcal{N}(C(E^2)) \setminus \mathcal{N}(E^2)$  for which the projection criterion  $\mathcal{S}(\psi_{x^1})$  fails. We then know by means of the saturation condition that  $\mathcal{S}(\psi_{x^2})$  also fails for all nodes  $x^2 \in \mathcal{N}(E^2)$ , especially for those on  $F^2 \subset F$ . Therefore  $\mathcal{S}(\psi_x)$  fails at least on one node  $x \in \mathcal{N}(C(\tilde{E}))$ . But this contradicts our assumption

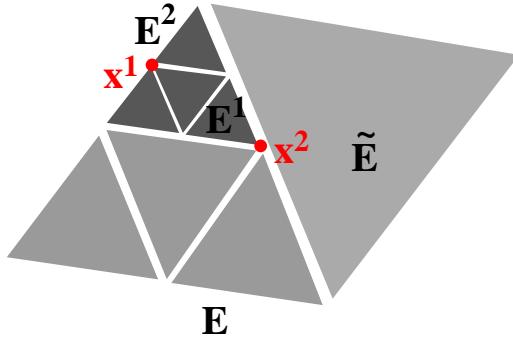


Figure 16: In the adaptive traversal at most one level transitions occur. Thus child element of  $E^2$  would not be visited, if the algorithm stops on  $\tilde{E}$ .

that  $S(\tilde{E})$  is true, once more because of the saturation condition.

In the case of general hierarchical function spaces a related result holds. If the diameter of the base function support decreases geometrically with increasing level, i. e. there exist constants  $C$  and  $\alpha$  such that  $\text{diam}(\text{supp}(\phi_i^m)) \leq C \alpha^m$  for  $\phi_i^m \in \mathcal{B}_+^m$  and the grid hierarchy is regular in the usual sense, then the grid size function  $h(x)$  is a Lipschitz continuous function. Here  $h(\cdot)$  on the nodes is defined as the maximal diameter of adjacent elements and as the corresponding linear interpolation in between.

Up to now we have constructed an interpolation  $P_{SU}$  of the original function, which is continuous and can efficiently be computed by an interactive visualization method. But nevertheless, whenever the hierarchical grid on which we resample discrete function data consists of elements with some bilinear faces (e. g. hexahedral or prismatic grids) we have to cope with a hidden algorithmical difficulty. If we confine with patches which are linear simplices in the extraction of isosurfaces, the resulting isosurfaces or isolines on slices might have cracks. On a face  $F$  with a transition in the adaptive element levels the interpolation is continuous whereas on the finer elements on one side we will in general generate a better approximation of the bilinear function representing the projection on the face  $F$  than on the coarser element on the other side (cf. Fig. 17). That would lead to a small crack. In the case of isolines on slices cracks in the lines occur only if the clipping plane is not aligned to the principle direction of the grid. On 2D domains or surfaces this difficulty is ruled out. Concerning the 3D multilinear finite element case we already know that the algorithm generates at most one level transitions at faces of elements which fulfill the stopping criteria. Thereby the crack problem can easily be solved. On an element all rectangular faces pointing to finer adjacent elements have to be subdivided once before the local isosurface is being extracted. This ensures identical discretizations for the considered faces from both sides.

Figure 17: A crack at a face on which the function is bilinear and a level transition occurs. The face is extracted and the intersections of the isosurface from both sides are drawn as a dotted and a dashed line, respectively. Further lines indicate the subdivision on one element faces.

## 7 Efficient Hierarchical Searching on arbitrary Nested Grids

Post processing in computational fluid dynamics requires efficient visualization on unstructured, hierarchical and often economical stored grids. Particle methods which generate and visualize particle lines, stream surfaces, or moving clouds of particles are important tools for the examination of velocity fields. The efficiency of these methods depends sensitively on searching algorithms. An ODE solver like any Runge Kutta scheme requires the evaluation of numerical function data at successively calculated points on a particle line. For this evaluation the element location and the local coordinates of these points have to be computed in advance. Therefore searching and especially local searching in the grid hierarchy turns out to be a fundamental task. Fig. 18 displays particle straces, a streamsurface and some timesteps of the evolution of a ball shaped surface in the flow.

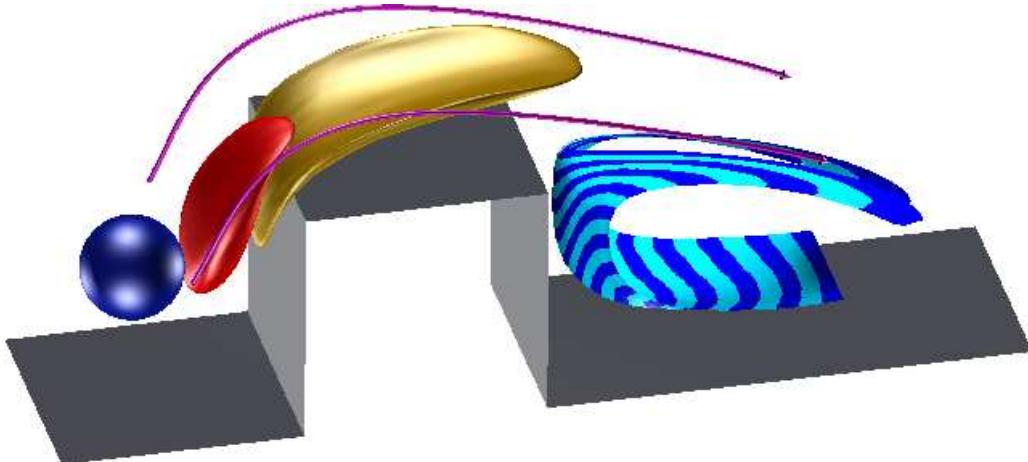


Figure 18: Application of particle tracing methods to a numerical solution of the Navier Stokes equations.

Especially for economically stored, very large data sets the storing of adjacency information to support local searching is ruled out. For hierarchical meshes consisting of elements which are cartesian products of simplices and recursively generated applying a finite set of refinement rules, a different efficient method for hierarchical searching is available. Let us here briefly summarize the basic idea.

Depending on the refinement rule the conversion of local coordinates on child elements to local coordinates on a parent element can be coded as linear mappings in a preparatory step. The same holds for the inverse mapping. Frequently these linear maps are rather simple. Not every new image coordinate depends on all preimage coordinates and many of the resulting single component maps are linear dependent. If we use these relations in the pre computing step and sort the child elements appropriately an efficient hierarchical searching can be implemented. In terms of computational effort this strategy is almost optimal (Cf. Section 12 for implementational hints).

## 8 Concept for the Procedural Data Access

Up to now we have encountered a variety of different numerical data structures and developed a general multiresolutional strategy for data in corresponding hierarchical function spaces. A derived visualization algorithm consists of a recursive traversal of the grid hierarchy, checking for error indicators, evaluation of the hierarchical offsets, and finally some local rendering. Let us now consider a general interface to numerical data, which covers all these different classes of data.

At first we might ask for a general data structure and file format, to which data has to be converted. But such a global, general format would be storage expensive and not be very handsome. Especially the representation of very different local function spaces in prescribed data structures is crucial. Beyond this it would be desirable to have an easy and fast feed back with the numerical method, which would already help to debug the numerical algorithm.

A procedural interface which

- directly accesses the user's data structures,
- supports hierarchical grid traversal,
- delivers information on element geometry only temporarily and locally,
- represents local function spaces by user supplied function procedures and
- includes local procedures which evaluate error indicators

and those procedures are filled by the user overcomes the drawbacks of data formats.

An hierarchical element  $E$  on any grid  $\mathcal{M}^l$  of level  $l$  is described by some structure  $HELEMENTnD$ , where the  $n$  indicates the dimension in space two, respectively three. This structure especially contains a reference to some  $HELEMENTnD\_DESCRIPTION$  and the coordinate vectors for the vertices.

The  $HELEMENTnD\_DESCRIPTION$  consists of a polygon oriented boundary specification, where each face is represented by a list of vertex indices. Furthermore linear local coordinate spaces, on tensor product elements typically the tensor products of barycentric coordinates on simplices, are added as black boxes. User supplied procedures describe the mappings between world coordinates and local coordinates and thereby allow the handling of these black boxes.

The local *function*  $f$  naturally lives in the local coordinate space. Tailored to the numerical data it can easily be evaluated in these coordinates. The user is required to provide a corresponding function procedure in the interface (cf. Section 2). In a corresponding data structure  $F\_HDATAnD$  this function, a procedure which evaluates the function related stopping criteria and a procedure which computes lower and upper bounds for the local function on elements (typically based on error indicator information) is supplied.

The hierarchical mesh (denoted  $HMEShnD$ ) itself, consists of a reference to the *function* and procedures, which build up a procedurally linked graph of the corresponding  $HELEMENTnD$  hierarchy (cf. Fig. 19). Two procedures *first\_macro()*, *next\_macro()* successively deliver information on the coarse grid elements in the  $HELEMENTnD$  data structure, overwriting previous element data. A call of *first\_child()* generates and fills an additional  $HELEMENTnD$  structure with some first child data. Finally successive calls of *next\_child()* traverse the other child elements of the same parent element and replace previous child data (cf. Fig. 19). Thereby during a recursive traversal of the grid hierarchy, a hierarchical, backwards linked list of at most  $m$  temporarily filled  $HELEMENTnD$  structures is present in memory at the same time, where  $m$  is the depth of the hierarchy. This especially implies that also in the user's 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. Again the user has to supply or adapt these procedures corresponding to the specific numerical data.

Finally additional procedures support other types of visualization methods. A procedure *select\_child()* works similar to *first\_child()*, but provides one specific child which contains a point given in local coordinates. Together with a procedure which maps local child coordinates on an element  $E$  to local parent coordinates on  $\mathcal{P}(E)$  this allows an efficient recursive implementation of local searching for particle tracing methods.

Furthermore local element adjacency can also be evaluated by a procedure *neighbour()* which replaces the  $HELEMENTnD$  list for some hierarchical element by the corresponding list for a neighbouring element. This routine is not required to be filled (cf. Section 7).

Let us at first summarize the main properties of this approach. In general at the same time only a small number of grid elements, given in prescribed  $HELEMENTnD$  structures is present in memory. There is no random access to a single element. The above access procedures supply a hierarchical visualization method with all information necessary to locally evaluate and graphically represent grid geometry and data. This is sufficient to run nearly all visualization algorithms, e. g. isosurface rendering, isoline drawing, data dependent color shading, volume rendering and particle tracing. 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. Hierarchical grid traversal and hierarchical searching are efficiently supported. The user only has to provide the interface procedures and to give a description of the element types. In most cases existing interfaces can easily be adapted to new numerical data structures, which are thereby accessible for efficient multilevel visualization methods.

## 9 Handling Ecomonically Stored Very Large Data

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

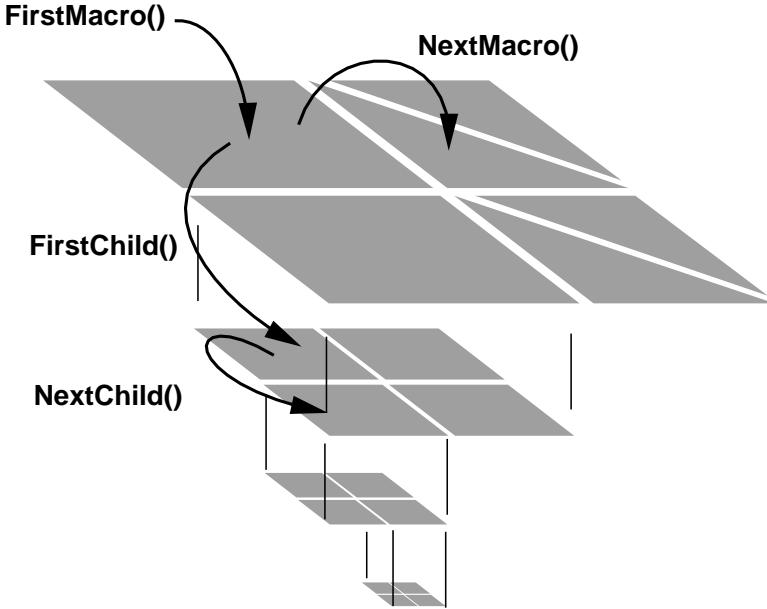


Figure 19: A schematic sketch of the procedural access to hierarchical grids in 2D by the four routines *first\_macro()*, *next\_macro()*, *first\_child()*, *next\_child()*.

to address a hierarchical grid procedurally for numerical or graphical purposes. We discuss here the case of multilinear Finite Elements on recursively refined nested grids.

For each element of such a grid 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  $\mathcal{N}_c(E) \setminus \mathcal{N}(E)$  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.

<b>MacroElement</b>		<b>EcoElement</b>	
<b>MacroNode</b>	*macroNode	<b>int</b>	refrule
<b>MacroElement</b>	*neighbour	<b>Node</b>	**newNode
<b>EcoElement</b>	*self	<b>BoundPoint</b>	*boundPoint
		<b>EcoElement</b>	*child

Figure 20: 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)

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. 20). Let us mention that the highest level elements 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 *HELEMENTnD* structure (possibly slightly enlarged by some additional user data) in a preceding step of the recursion (cf. Sect. 13). 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.

## **Part II      Implementational Aspects**

## 10 Description of the Interface Structures

The base structure implementing the multiresolution concepts is `HMesh3d` and `HMesh2d` for three and two space dimensions respectively. For the description of the implementational aspects we will always restrict to either the three dimensional or the two dimensional case, as the concepts for both situations are transferable. The hierarchical concept (`HMesh3d`) is implemented as a subclass of `Mesh3d`. Additionally to `Mesh3d`, which deals with non-hierarchical meshes, (`HMesh3d`) is enlarged by the hierarchical element functionality and expanded interface structures. Although `MESH3D` has already been described in an earlier paper [34], its documentation is in revised form listed once more for the sake of completeness and with respect to actualizations.

### Hierarchical Element

In this mesh concept we use a procedural access to single elements. The procedures return element data in a small structure, these procedures are described later on. One example is for instance

```
HELEMENT3D *first_macro(HMESH3D *mesh, MESH_ELEMENT_FLAGS required);
```

This — user supplied — function has to return the first macro element of a given `HMesh3d`. The flags `required` specify, which parts of the `HELEMENT3D`-structure are to be filled at least.

```
typedef struct helement3d
{
    MESH3D           *mesh;
    double          **vertex;
    int              *vindex;
    int              eindex;
    ELEMENT3D_DESCRIPTION *descr;
    MESH_ELEMENT_FLAGS   present;
    void             *user_data;
    HELEMENT3D        *parent;
    VINHERIT         *vinh;
    int               ref_rule;
    int               level;

} HELEMENT3D;
```

The structure for an element of the mesh consists mainly of a pointer `descr` to an element description, which contains all information described above, and a vector of pointers `vertex` to the world coordinates of the element's vertices.

Such information may be enlarged by optional global `eindex` and `vindex`[] integer indices of the element itself and its vertices and a pointer to a data area which contains additional information that may be used by the transformation routines or any other of the user-supplied routines.

The vertex indices, if present, allow the visualization routines to minimize the number of calls to the data function, if the data values are globally continuous. In this case, the values at a node are uniquely defined, independent of the element where it belongs to. If the data is not continuous between elements, the global vertex index may also be used to compute a continuous approximation to the data by averaging the values at the nodes.

The global element indices for example may be used to specify a set of elements of special attention and to display only these elements (or their neighbourhood).

The flag variable `present` shows, which data arrays are set, at least the ones, which were set in the function call. The pointer `parent` points to the `HELEMENT`-structure of the parent element, if there is one. This allows a flexible handling of the existing list of element, from the coarsest parent element to the current decendent element.

Each of the later described funtions has to set the `mesh`, `descr`, `present` and `parent` entries. Since `mesh` is an `HMesh` in this context is must of course be downcasted. If the `max_eindex` entry of `mesh` is nonzero, `eindex` must be set to the element's global number, else to zero. `ref_rule` is completely optional. The optional `user_data`

can point to additional data and is not dereferenced in the library. `vertex`, `vindex` and `vinh` only need to be set, if the `required` flags of the function call says so. `vertex` points to an array of pointers, pointing to an array containing the world coordinates of the vertices. If the `max_vindex` entry of `mesh` is nonzero, `vindex` must point to an array of the vertex's global numbers, else it must be `NULL`. `vinh` points to an array of `VINHERIT` structures

for each vertex, or is `NULL`.

`descr` points to an element description, which is constant for a whole class of elements, and which will be described hereafter:

The geometry of a single element is described using a polygon-oriented boundary representation of the parameter domain. As we support only (curved) polyhedral elements, an element can be described by the set of its vertices and a description of the boundary polygons. This applies only to the local coordinates of an element; its shape in world coordinates is determined by a transformation routine.

For a three dimensional polyhedron, we specify the number of boundary polygons (faces) and for each of these polygons the number of vertices, the local vertex indices and their order (thus giving an orientation to the polygon such that the surface normal is the inner normal to the polyhedron), and the local indices of the adjacent face across each of the edges of the polygon.

For each vertex, the coordinates in the local coordinate system of the element are given by the element description. Based on these values, the visualization routines may operate in the local coordinate space. The pointer to a function `check_inside()` is provided by the element description, which checks whether a point in local coordinates is inside the element or not.

The element description is completed by pointers to transformation routines from local coordinates to world coordinates and vice versa and routines `boundary()` and `neighbour()`, which give information about the neighbourhood of elements. Additionally, the transformation routine from world to local coordinates checks whether a given point is inside the element or not.

```
typedef struct helement3d_description
{
    int          dindex;
    int          number_of_vertices;
    int          number_of_polygons;
    int          *polygon_length;
    int          **polygon_vertex;
    int          **polygon_neighbour;
    int          dimension_of_coord;
    double       **coord;
    int          parametric_degree;
    int          (*world_to_coord)(HELEMENT3D *, double *, double *);
    void         (*coord_to_world)(HELEMENT3D *, double *, double *);
    int          (*check_inside) (HELEMENT3D *, double *);

    HELEMENT3D  *(*neighbour)(HELEMENT3D *, int, int, double *, double *, MESH_ELEMENT_FLAGS);
    int          (*boundary) (HELEMENT3D *, int);

    void         (*get_boundary_vertex_estimate)(HELEMENT3D *, double *, int );
    double       (*get_boundary_face_estimate) (HELEMENT3D *, int );
    void         (*coord_of_parent)           (HELEMENT3D *, double *, double *);
};

} HELEMENT3D_DESCRIPTION;
```

The entries `dindex` to `boundary` are inherited from `HMesh3d`. The variables give following information:

`dindex` is an index of the current description. This is to distinguish descriptions, which might be processed in a distributed environment.

`number_of_vertices` is the number of vertices in an element of this type.

`number_of_polygons` is the number of polygonal faces of an element of this type.

`polygon_length[j]` is the count of vertices of the  $j$ -th face ( $0 \leq j < \text{number\_of\_polygons}$ ).

`polygon_vertex[j][i]` is the (local) index (with range  $[0, \dots, \text{number\_of\_vertices} - 1]$ ) of the  $i$ -th vertex from the  $j$ -th face ( $0 \leq j < \text{number\_of\_polygons}$ ,  $0 \leq i < \text{polygon\_length}[j]$ ).

`polygon_neighbour[j][i]` is the index of the adjacent face to the  $j$ -th face across the edge from vertex `polygon_vertex[j][i]` to vertex `polygon_vertex[j][i + 1]` ( $0 \leq j < \text{number\_of\_polygons}$ ,  $0 \leq i < \text{polygon\_length}[j]$ ).

`dimension_of_coord` is the dimension of the local coordinates.

`coord[k][i]` is the  $i$ -th local coordinate of the  $k$ -th vertex from the polyhedron ( $0 \leq i < \text{dimension\_of\_coord}$ ,  $0 \leq k < \text{number\_of\_vertices}$ ).

`parametric_degree` is the degree of parametric elements.

`check_inside(element3d, coord)` returns `INSIDE` (`-1`) if the point with local coordinates `coord` is inside of the element's parameter range. Otherwise, at least one plane of a local face separates the element and the given point. In this case the local index of such a face is returned.

`world_to_coord(element3d, xyz, coord)` transforms a point from given world coordinates `xyz[3]` to `coord[dimension_of_coords]` in the local coordinate system of `element3d`. The return value is identical to that from `check_inside()`.

`coord_to_world(element3d, coord, xyz)` transforms a point from local coordinates `coord[dimension_of_coords]` to world coordinates `xyz[3]`.

`neighbour(element3d, polygon, flag, coord, xyz, required)` returns a pointer to an adjacent element of `element3d` across face number `polygon`. Depending on `flag`, the routine returns the first neighbour or a next one (in case of non-conforming meshes the adjacency may be not one to one). If `coord` is not `NULL`, the routine transforms those coordinates from the old local coordinate system to the new one. In many cases this transformation can be supported by the optional parameter `xyz`, which provides world coordinates of the point if available, otherwise this parameter has to be `NULL`. The structure `element3d` may be overwritten by the routine. On hierarchical data not explicitly storing neighbour pointers, it is a bad idea to implement this function, set its pointer to `NULL` to mark this.

`boundary(element3d, polygon)` returns 0, if `polygon` is an inner face of the mesh, a negative integer, if `polygon` is part of the mesh's outer boundary and a positive integer for interior boundary layers.

`get_boundary_vertex_estimate(element3d, estimate, face)` determines the geometry boundary estimator for vertices lying on face `face` of `element3d`, using the current level. Cf. sections 4, 13.8 for details.

`get_boundary_face_estimate(element3d, face)` returns the geometry boundary estimator for face `face` of `element3d`, taking into account the finer level. Cf. sections 4, 13.8 for details.

`coord_of_parent(element3d, child_coord, parent_coord)` determines the local coordinates of a point, given in `element3d`'s local coordinates `child_coord`, in its parent's local coordinates `parent_coord`. Since this can be implemented in a less efficient way using the above functions, there is a default function for this purpose, named `g_hmesh3d_def_coord_of_parent`.

The number of such element descriptions and therefore the amount of storage for them is equal to the number of different element types in one mesh. There are no copies of this necessary for each element.

In Figure 21 vertex and face relationships for the example of cubes are depicted. `check_inside()` is easily implemented using the local coordinates.

The user's data structures (e.g. from a numerical method) may be completely different to the structures described above. In case of implicitly given Finite Difference meshes, the functions `neighbour` and `boundary` are implemented using simple index arithmetic.

Unstructured grids usually provide such information explicitly.

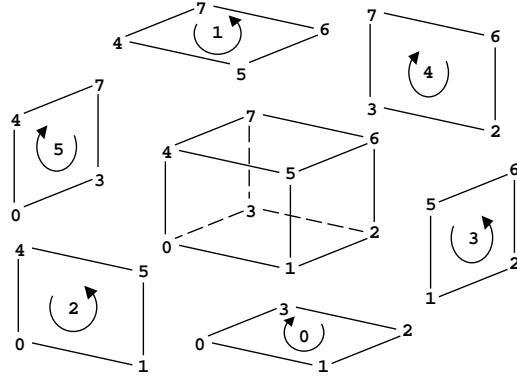


Figure 21: Prototype of a cube.

```
typedef struct vinherit
{
    int      np;
    int      *pindex;
    double   *pweight;

} VINHERIT;
```

This structure defines optional vertex inheritance for a vertex. If the required flags order a `vinh` array and none can be given, it has to be set to `NULL`. Ist entries are

`np` number of parent vertices

`pindex` local indices of parent vertices

`pweight` weights for the coordinate calculation

A vertex also appearing in the parent element would have a `VINHERIT` of `{1,{i},{1.0}}`, one being interpolated between two vertices  $i$  and  $j$  by bisection would have `{2,{i,j},{0.5,0.5}}`.

## Supervising Grid Structure

Now we introduce the supervising structures of the actual tree of elements and the function data defined on the corresponding domain. At first we introduce an enumeration type which will serve as a parameter type to describe the required information of a given element for a specific visualization task.

```
typedef enum
{
    hefNone     = 0,
    hefVertex   = 1<<0,
    hefVindex   = 1<<1,
    hefVinh    = 1<<2,
    hefAll      = 1<<0 | 1<<1 | 1<<2

} MESH_ELEMENT_FLAGS;
```

More precise `hefNone` indicates no information except the basic one is to be specified, `hefVertex` marks that vertex coordinates have to be supported. Vertex indices (if present) have to be delivered, if `hefVindex` is set, and finally coordinate inheritance rules have to be supported whenever `hefVinh` is marked.

In what follows the class description for nonhierarchical meshes `MESH3D` and for hierarchical meshes `H_MESH3D` are listed.

```

typedef struct mesh3d
{
    int             max_dimension_of_coord;
    int             max_eindex;
    int             max_vindex;
    int             max_dindex;
    int             max_number_of_vertices;

    G_LIST          *current_function; /* private */

    MESH_ACCESS_FLAGS access_mode;
    MESH_ACCESS_FLAGS access_capability;

    GENMESH_FDATA   *f_data;

    void            *user_data;

    ELEMENT3D      *(*first_element)(MESH3D *, MESH_ELEMENT_FLAGS);
    ELEMENT3D      *(*next_element) (ELEMENT3D *, MESH_ELEMENT_FLAGS);
    ELEMENT3D      *(*copy_element) (ELEMENT3D *, MESH_ELEMENT_FLAGS);
    void           (*free_element) (ELEMENT3D *);

    ELEMENT3D      *(*complete_element)(ELEMENT3D *, MESH_ELEMENT_FLAGS);

    int             (*set_time)(MESH3D *, double t);
    int             (*get_time)(MESH3D *, double *t, double *tminus, double *tplus);

} MESH3D;

```

This mesh class is a superclass of the following class `hmesh3d`:

```

typedef struct hmesh3d
{
    int             max_dimension_of_coord;
    int             max_eindex;
    int             max_vindex;
    int             max_dindex;
    int             max_number_of_vertices;

    G_LIST          *current_function; /* private */

    MESH_ACCESS_FLAGS access_mode;
    MESH_ACCESS_FLAGS access_capability;

    GENMESH_FDATA   *f_data;

    void            *user_data;

    ELEMENT3D      *(*first_element)(MESH3D *, MESH_ELEMENT_FLAGS);
    ELEMENT3D      *(*next_element) (ELEMENT3D *, MESH_ELEMENT_FLAGS);
    ELEMENT3D      *(*copy_element) (ELEMENT3D *, MESH_ELEMENT_FLAGS);
    void           (*free_element) (ELEMENT3D *);

    ELEMENT3D      *(*complete_element)(ELEMENT3D *, MESH_ELEMENT_FLAGS);

```

```

int          (*set_time)(MESH3D *, double t);
int          (*get_time)(MESH3D *, double *t, double *tminus, double *tpplus);

HELEMENT3D   *(*first_child) (HELEMENT3D *, MESH_ELEMENT_FLAGS);
HELEMENT3D   *(*next_child)  (HELEMENT3D *, MESH_ELEMENT_FLAGS);
HELEMENT3D   *(*first_macro) (HMESH3D *,      MESH_ELEMENT_FLAGS);
HELEMENT3D   *(*next_macro)  (HELEMENT3D *, MESH_ELEMENT_FLAGS);
HELEMENT3D   *(*select_child)(HELEMENT3D *, double *, double *, MESH_ELEMENT_FLAGS);
int          max_level;
int          level_of_interest;
double       boundary_threshold;

} HMESH3D;

```

The mesh is just a collection of elements, as they are described in the previous section, together with optional data functions on the mesh. As already mentioned Mesh3d only contains the subset of instance variables from `max_dimension_of_coord` to `get_time`.

We restrict the access to elements to a procedural interface, that delivers pointers to elements in one of two possible ways:

- by building a procedurally linked list of elements, that runs sequentially through all elements in the mesh via two procedures

```

ELEMENT *first_element(mesh,required),
ELEMENT *next_element(element,required),

```

which return the anchor to the list resp. the next element in the list, having to provide all data not mentioned in `MESH_ELEMENT_FLAGS` and all arrays marked in `required`. As shown later on, for an HMesh3d these functions can be implemented using their hierarchical counterparts. Thus default functions are present in new instances.

- we can go from one element to an adjacent element (in space) across the face polygon via the routine

```

ELEMENT *neighbour(element, polygon, flag, coord, xyz, required)

```

which is contained in the `HELEMENT3D_DESCRIPTION`.

Both the `next_element()` and `neighbour()` routines may overwrite the `element` data structure such that no additional storage is needed for the next element's data structures. In case, where one needs to collect information about several elements, a routine

```

ELEMENT *copy_element(element,required)

```

gives a copy of an element, which can be deleted later by

```

free_element(element,required).

```

When an element lacks some information, in conformance to its `present` flags, there may be called

```

ELEMENT *complete_element(element,required)

```

to complete it. Normally it would return the same, updated, element again; of course with correct `present`. The hierarchical extensions/replacements of `first_element` and `next_element` are

```

ELEMENT *first_macro(mesh,required),
ELEMENT *next_macro(element,required),

```

which traverse the macro elements in an equivalent way and

```

ELEMENT *first_child(parent,required),

```

```
ELEMENT *next_child(element,required),
```

where `first_child` returns the first child of `parent` on the next finer level, and subsequent calls of `next_child` traverse the other children on this level.

```
ELEMENT *select_child(parent,parent_coord,child_coord,required)
```

is only optional and needed for efficient hierarchical searching. If the point with the local coordinates `parent.coord` is contained in a child of `parent`, a new element is created, filled with the child's information, and returned. In addition the local coordinates of the point are converted to the child's local coordinate system and returned in `child.coord`. There is a default function, `g_hmesh3d_def_select_child`, for this using the above described functions, but in many situations there exists a direct access to the children, which results in a speedup.

The variables `access_mode` and `access_capability` indicate the currently set traversal mode and the supported traversal modes respectively. It is the user's responsibility to set the flag `access_capability` and to program the mesh traversal routines (`first_element`, `next_element`, etc.) so that the set access capabilities are indeed supported. At the moment there are only two possible access flags for different types of mesh traversal (c.f. Section 14.6.3 for further explanations:

```
typedef enum {
    mafNone      = 0,
    mafSorted    = 1<<0,
    mafBoundary  = 1<<1
} MESH_ACCESS_FLAGS;
```

The integer `max_dindex` gives an upper bound of the used element description indices.

This procedural access to the mesh elements allows the generation of local element data at the time when it is actually needed. Otherwise, in case of an array or pointed list, complete information for all elements of the mesh would have to be present at the same time (and occupy much more storage).

The routines which generate the element data structures have to convert between the representations via user's data structures and the element structures. They do this mainly by assigning vertex and index information for an element and referring to a previously prepared description. As mentioned above, in case of structured meshes this is done mainly by index arithmetic, whereas for unstructured meshes such information is usually stored with the mesh.

The `MESH3D` data structure contains values for the allocation of temporary memory that some routines may need:

- An upper bound `max_number_of_vertices` for the number of vertices of any element in the mesh such that a routine which wants to store local vertex information knows how much memory it has to allocate.
- Upper bounds `max_eindex` and `max_vindex` for the element index and vertex index, if such information is supplied (then the `eindex` and `vindex[]` entries in the `ELEMENT` data structure deliver values in the range  $[0, \dots, \text{max\_eindex}-1]$  resp.  $[0, \dots, \text{max\_vindex}-1]$ ). If any one of the `max_?index` values is smaller or equal to zero, the elements do not give corresponding index information.

Furthermore the mesh supports function evaluation in an abstract and flexible way. There can be defined several functions on the mesh. They are all independent concerning properties such as continuity and polynomial degree. The entry `current_function` is a pointer to a list of currently supported functions on the mesh. The `GLIST` is a simple list supervising class type.

The functions `set_time()` and `get_time()` enable the handling of timedeependent data. Calling `set_time()` with time `t` ensures that the next function evaluation is with respect to this new time. Furthermore, meshes deforming in time, will then – if supported – deliver time dependent coordinate positions. Finally, the access routines consider the correct adaptive grid valid for the specified time, with its uniquely defined tree hierarchy. The call of `get_time()` returns the current time `*t` and the current time interval `[*tminus,*tplus]`. In case of adaptive grids changing their grid topology in time, the identification of an time interval is very useful to be sure that the grid is constant on this time interval.

An `HMESS3D` has the additional variables

**max\_level:** The maximal hierarchy level, e. g. 0 for an HMesh3d only consisting of macro elements, 1 if every macro element has children, but no grandchildren, etc.

**level\_of\_interest:** The maximal hierarchy level that is of interest right now. Recursive traversal of the elements will stop at this level; which is a simple way to reduce computing times.

**boundary\_threshold:** The threshold for the error on displaying the boundaries.

Due to their nature, **level\_of\_interest** and **boundary\_threshold** can be adjusted interactively at the graphical user interface.

At last, the HMESH3D structure contains an interface to data given *on* the mesh. Here we want to support a situation where different types of data are given at the same time on the same mesh with different value dimensions and other characteristics (example: a piecewise linear, scalar pressure, a piecewise quadratic, vector-valued velocity, and some more), together with some methods to select one of the data values for display. The data characteristics may change between two visualization tasks. Additionally, there may be data characteristics which change from element to element in the same mesh, for example the polynomial degree of the basis functions. To handle all these situations, we use a rather general interface for functions on the mesh. A user supplied procedure evaluates data selected for visualization. Instead of only supporting functions given in terms of a Lagrangian basis, this concept is open for any type of user function such as one of Hermitian type or one only piecewisely defined on several parts of a single element, for example.

Data to be displayed is defined globally on the whole mesh, although the evaluation of data is done on element level (using local coordinates). Thus, the data interface is located at mesh level and described by the following structure:

## Discrete Functions

```
typedef struct f_hdata3d
{
    char          *name;
    GENMESH_FDATA *last, *next;
    int           dimension_of_value;
    int           continuous_data;
    void          *function_data;

    void          (*f)      (ELEMENT3D *, int, double[], double[], void *function_data);
    void          (*f_el_info)(ELEMENT3D *, F_EL_INFO3D *,           void *function_data);

    void          (*get_bounds)(HELEMENT3D *, double *, double *, void *function_data);
    void          (*get_vertex_estimate)(HELEMENT3D *, double *, void *function_data);
    double        (*get_element_estimate)(HELEMENT3D *,           void *function_data);
    double        threshold;
    double        geometry_threshold;
    double        hp_threshold;
    int          hp_maxlevel;

} F_HDATA3D;
```

This is the structure defining a function on an hierarchical mesh. The entries **name** to **f\_el\_info** are inherited from Mesh3d, which is again a subclass of Gen\_Mesh; due to this inheritance the F\_HDATA3D has to be downcast to GENMESH\_FDATA3D in assignments. All entries have to be set by the interface. **threshold**, **geometry\_threshold**, **hp\_threshold** and **hp\_maxlevel** can later be edited on the graphical user interface.

In detail, the entries are:

**name:** a textual description of current data (“pressure”, e.g.), internally used as the name of the function that will appear in selection dialogs on the user interface,

`dimension_of_value`: is the dimension of the function values, i. e. 1 for a scalar function, > 1 for a vector valued function,

`continuous_data`: contains the boolean (TRUE or FALSE) information, if the function is continuous between elements,

`f(el, vindex, coord, val, function_data)`: the data function itself generating `dimension_of_value` – dimensional values at a position which has to be specified in local coordinates on element `el`. This function may also be called with parameters `coord=NULL` and a local index `vindex` of one of the element's vertices.

`f_el_info(el, el_info, function_data)`: fills structure `el_info` with local information about data, such as polynomial degree for use by routines which adaptively choose the display resolution, e.g.

`last, next`: to build a doubly linked list of data descriptions.

`function_data`: optional pointer, passed on every function call, in which the user can store information needed to calculate the function values or to identify the function in the case of dynamical allocation.

`get_bounds(el, min, max, function_data)`: calculates the lower- and upper bound of the function values in an element. If this cannot be determined in an efficient way, return  $-\infty$  and  $+\infty$ .

`get_vertex_estimate (el, vertex_est, function_data)`: calculates the error estimator for all vertices of `el`, filling `vertex_est` with these values.

`get_element_estimate (el, function_data)`: returns the error estimator for the children, thus determining, if the element is to be refined.

`threshold, geometry_threshold, hp_threshold`: are the thresholds for function, geometry and on-element subdivision in case of higher polynomial degrees, respectively. If an error indicator is above its threshold, refinement will be attempted.

`hp_maxlevel`: is the maximal refinement level for on–element subdivision in case of higher polynomial degrees. Its name resembles *hp-methods*.

The `f_data` entry in the `HMESH3D` structure gives access to data which is currently selected for display. By use of the doubly linked list, an interactive choice between different data is possible.

In the end, the `HMESH3D` structure contains a data pointer `user_data`. This simplifies the simultaneous handling of multiple mesh3d data sets, if for any reason one does not want to create **GRAPE** subclasses of `HMesh3d`.

```
typedef struct f_el_info3d { int polynomial_degree; } F_HEL_INFO3D;
```

The structure, `f_el_info()` has to fill with information about the function. Currently the only entry is `polynomial_degree`, which is the function's polynomial degree on the elements, i. e. 1 for a piecewise linear function.

## 11 Writing Display Methods for the Classes HMesh2d and HMesh3d

### 11.1 Introduction

The GRAPE-library contains a large variety of methods for visualizing data given on hierarchical grids. On the one hand there are methods for analysing the grid itself on the other hand functions defined on the grid can be presented in different ways. However, not all possible methods and algorithms for extracting and visualizing data can be included in advance. So in accordance with the open and extensible concept of GRAPE the user can write his own display methods. How to do this and which tools exist in GRAPE to facilitate this is the scope of the following sections.

In general any display method will have the following structure:

- Preparation, includes getting the object to be displayed, getting a graphic device, initializing interactive elements with which parameters of the display method can be altered and possibly getting additional tools as for example the colorbar.
- Data extraction, by means of the interface functions of the Hmesh – first-macro, next-macro, etc. – and the access function of the F\_hdata-structure the necessary data is extracted. As this process is very similar in many display methods there are powerfull tools for the most common extraction requirements.
- Rendering, finally using the functionality of the graphic device the extracted data are rendered. This part of the display method is almost always specific for it and thus it is the part where except for the routines provided in the graphic device no other tools are available.

In the following each of the steps mentioned above is discussed in detail. Once the user has written a display-method it must be added to the appropriate class as described in [38]. In order to be able to choose the display method in the manager a naming convention must be complied with.

Most examples in this section are based on the class **HMesh2d**. However, the corresponding class **HMesh3d** is similar and most of the examples work also in the 3D-case.

## 11.2 Preparation

The task of the preparational part of the display method is to provide access to the necessary objects and data structures. As an example the beginning of the hmesh2d isoline display method is printed below. In addition to the **HMesh2d** object the isoline method needs access to the function values. Therefore a pointer to a **F\_HDATA2D** structure is initialized. As the isolines are colored according to their iso-value a colorbar is neccessary. Besides the obligatory graphic device the use of a lens is prepared. For more details concerning the different objects see their respective documentation.

```
HMESH2D* hmesh2d_isoline_disp(void)
{
    HMESH2D*      hmesh;
    F_HDATA2D*    f_hdata2d;
    COLORBAR*     colorbar;
    GRAPHICDEVICE* device;

    static HMESH2D_LENS* lens = NULL;

    /* get hmesh and function */

    hmesh = (HMESH2D*)START_METHOD(G_INSTANCE);
    ALERT(hmesh,"hmesh2d-isoline: No hmesh!",END_METHOD(NULL));

    f_hdata2d = (F_HDATA2D*)GRAPE(hmesh,"get-function")
        ("scalar","scalar","vector","default",NULL);
    ALERT(f_hdata2d,"hmesh2d-isoline-select: No function!",
        END_METHOD(NULL));

    /* get colorbar and graphicdevice */

    colorbar = (COLORBAR*)GRAPE(Colorbar,"get-stdcolorbar")
        (hmesh2d_isoline_disp,"isoline-disp");

    device = (GRAPHICDEVICE*)GRAPE(GraphicDevice,"get-stddev")();

    /* get lens */

    if(!lens)
```

```

lens = (HMESH2D_LENS*)GRAPE(Hmesh2d_Lens,"get-lens")
("lens for hmesh2d-isoline");

...
}

```

Often the visualization algorithms implemented in display methods depend on parameters that allow the user to manipulate the resulting picture. To make this possible interactive elements must be added to the manager. Below an excerpt of the display method hmesh2d-shrink-disp is printed. This method has one parameter which controls the size of the shrunk elements.

```

HMESH2D* hmesh2d_shrink_disp(void)
{
    MANAGER* mgr;
    ...
    static GROUP* group          = NULL;
    static SLIDER* shrink_size_ctl = NULL;
    static double shrink_size     = 0.95;
    ...

    /* get manager */

    mgr = (MANAGER*)GRAPE(Manager,"get-stdmgr")();

    /* get interactives */

    if(!shrink_size_ctl)
        shrink_size_ctl =
            (SLIDER*)new_item(Slider,
                I_Label,"shrink-factor",
                I_Var,&shrink_size,dfDouble,
                I_MinMax,0.0,1.0,
                I_Scale, 1.0,
                I_Offset,0.0,
                I_RSizeX,0.5,I_SizeY,1.0,
                I_End);

    if(!group)
        group = (GROUP*)new_item(Group,
            I_Border,bfBorder|bfTitle,
            I_Name,"hmesh2d-shrink",
            I_Size,12.0,1.25,
            I_Item,shrink_size_ctl,
            I_End);

    if(GRAPE(mgr,"new-handle")(hmesh2d_shrink_disp,1))
        GRAPE(mgr,"add-inter")(group);

    ...
}

```

## 11.3 Data Extraction

### 11.3.1 Overview

In this sections the tools that are helpfull for extracting data are presented. Although the context is to write display methods the tools are also usefull for calculating error estimators, norms of functions and for many other purposes.

There are exceptions, but most display methods extract the necessary data in the following way: The hierachical tree is traversed via the interface functions and on any leaf the data are fetched, calculated and rendered. As hanging nodes may occur in an adaptive traverse, hierachical projection is needed to assure continuity.

To support this, there are functions in GRAPE – namely `g_traverse_hmesh2d()` and `g_traverse_hmesh3d()` – that implement such an traverse. Among other things these routines need three call-back-functions as arguments.

- `projection`; in this routine the necessary data on the currrent element are collected and if appropriate interpolated.
- `test_if_proceed`; this functions controlls the traverse. It is decided whether to proceed on the next finer level or to stop on the current element and in the latter case whether to render on the current element or not. The outcome may be based on the projected data or for example on the error estimators included in the HMesh-interface.
- `action_on_element`; this functions should contain the code for rendering the data on an element.

Further arguments of `g_traverse_hmesh2d()` are the `Hmesh2d` object, a pointer to a `HM2_GENERAL` variable, the traverse mode, the level of interest, an array of pointers to the element data, a pointer to other arbitrary data possibly needed in the `action_on_element` routine and a mesh element flag.

The traverse mode, an variable of type integer, determines on which elements the `action_on_element` routine may be called. Possible values are `G_LEAVES`, `G_PREFIX` and `G_POSTFIX`. If the traverse mode is `G_LEAVES` the action is at most called on a leaf of the hierachical tree. In contrast if the traverse mode is `G_PREFIX` or `G_POSTFIX` the `action_on_element` routine may be called on all elements of the tree. In the case of `G_PREFIX` the `action_on_element` routine is called first on the parent element and after that the children are considered. Whereas in the case of `G_POSTFIX` only after the complete subtree of an element is traversed the `action_on_element` routine is called on the element itself.

The level of interest is the maximum depth of the hierachical traverse. In most display methods this should be the level of interest stored in the `Hmesh2d` object. Since this entry can be manipulated by the user via the uif-hm2-project it is not neccessary that display methods have an interactive element controlling it.

The interface functions of the `Hmesh2d` allow to choose which entries of the `HELEMENT2D` structure are filled by the interface. In the same way this choice can be made for the whole traverse with the mesh element flag.

Below the exact prototype of the traverse routine is printed. The other arguments are explained in the next section.

```
void g_traverse_hmesh2d(
    HMESH2D*      hmesh,
    HM2_GENERAL*  general,
    int           traverse_mode,
    int           level_of_interest,
    int     (*projection)      (HELEMENT2D* helement,
                               HM2_GENERAL* general,
                               void*       element_data,
                               void*       parent_data),
    int     (*test_if_proceed) (HELEMENT2D* helement,
                               HM2_GENERAL* general,
                               void*       element_data),
    int     (*action_on_element)(HELEMENT2D* helement,
                               HM2_GENERAL* general,
                               void*       element_data,
                               void*       action_arg),
    void**   element_data,
    void*    action_arg,
    MESH_ELEMENT_FLAGS mesh_element_flag)
```

### 11.3.2 Preparing the traverse

There are two different data structures used by the traverse functions. One structure is defined by the user and contains the element specific data. The other is of the type `HM2_GENERAL` and consists of pointers to objects and data structures that usually are needed in the `action_on_element`-routine but are not element specific, e.g. a pointer to the graphic device or the colorbar. The definition of this structure is given below.

```
typedef struct hm2_general{
    struct graphicdevice* dev;
    struct colorbar*      colorbar;
    struct hmesh2d_lens*   lens;
    F_HDATA2D*            f_hdata2d;
} HM2_GENERAL;
```

Note that it is not compulsory to fill all entries of this structure. If for example the `action_on_element`-routine does not need the colorbar the entry may be ignored by the user.

The user defined data structure with the element specific data may contain vertex coordinates, nodal function values or color values. The data structure used by the isoline display method for example employs all of the entries:

```
typedef struct hm2_coord_color_fct_data {
    VEC3    vertex_coord[MESH2D_MAX_VERTEX];
    VEC3    vertex_color[MESH2D_MAX_VERTEX];
    double  vertex_value[MESH2D_MAX_VERTEX];
    int     in_lens_area;
} HM2_COORD_COLOR_FCT_DATA;
```

Memory for an array of these structure variables must be allocated. The length of this array is given by the maximal depth of the hierachical tree. As an example we have again printed part of the display method `hmesh2d-shrink-disp`.

```
HMESH2D* hmesh2d_shrink_disp(void)
{
    ...
    HM2_GENERAL      general;
    HM2_COORD_DATA** hm2_shrink_data;

    int i,max;

    /* get hmesh, graphic device, ... and initialize interactives */
    ...
    /* prepare mesh-traverse */

    max = hmesh->level_of_interest + 1;

    general.dev  = dev;
    general.lens = lens;

    G_MEM_ALLOC(hm2_shrink_data,max);

    for(i=0;i<max;i++)
        G_MEM_ALLOC(hm2_shrink_data[i],1);

    /* traverse the mesh */

    g_traverse_hmesh2d(hmesh,
                        &general,
```

```

        G_LEAVE,
        hmesh->level_of_interest,
        hm2_coord_projection,
        hm2_geom_test_if_proceed,
        hm2_shrink_draw_element,
        (void**)hm2_shrink_data,
        NULL,
        hefVertex|hefVinh);

/* free memory */

for(i=0;i<max;i++)
    G_MEM_FREE(hm2_shrink_data[i],1);

G_MEM_FREE(hm2_shrink_data,max);

END_METHOD(hmesh);
}

```

The functions `hm2_coord_projection`, `hm2_geom_test_if_proceed` and `hm2_shrink_draw_element` must be defined somewhere else. The GRAPE-library contains several projection and test-if-proceed routines that can be reused and are explained in the next sections.

### 11.3.3 The projection routine

The purpose of the projection routine is to fill the element specific data in the element data structure. In the example above this was of the type `HM2_COORD_DATA` and contained only the vertex coordinates of the respective element. The data of the current element may depend on the data in the `HELEMENT2D` and the element data of the parent element. Additionally the entries of the `HM2_GENERAL`-structure may play a role in assembling and calculating the data on the current element.

The GRAPE library features several projection routines that can be reused by any other display method. In the 2D-case these are:

- `hm2_coord_simpl_projection`
- `hm2_coord_lens_projection`
- `hm2_scalar_fct_simpl_projection`
- `hm2_scalar_fct_lens_projection`
- `hm2_vec_fct_simpl_projection`
- `hm2_vec_fct_lens_projection`
- `hm2_coord_color_simpl_projection`
- `hm2_coord_color_lens_projection`
- `hm2_coord_color_fct_simpl_projection`
- `hm2_coord_color_fct_lens_projection`

These routines use the following data structure as element data:

```

typedef struct hm2_coord_data {
    VEC3 vertex_coord[MESH2D_MAX_VERTEX];
    int in_lens_area;
} HM2_COORD_DATA;

```

```

typedef struct hm2_scalar_fct_data {
    double vertex_value[MESH2D_MAX_VERTEX];
    int in_lens_area;
} HM2_SCALAR_FCT_DATA;

typedef struct hm2_vec_fct_data {
    VEC3 vertex_coord[MESH2D_MAX_VERTEX];
    VEC3 vertex_value[MESH2D_MAX_VERTEX];
    int in_lens_area;
} HM2_VEC_FCT_DATA;

typedef struct hm2_coord_color_data {
    VEC3 vertex_coord[MESH2D_MAX_VERTEX];
    VEC3 vertex_color[MESH2D_MAX_VERTEX];
    int in_lens_area;
} HM2_COORD_COLOR_DATA;

typedef struct hm2_coord_color_fct_data {
    VEC3 vertex_coord[MESH2D_MAX_VERTEX];
    VEC3 vertex_color[MESH2D_MAX_VERTEX];
    double vertex_value[MESH2D_MAX_VERTEX];
    int in_lens_area;
} HM2_COORD_COLOR_FCT_DATA;

```

The names of the projection routines indicate which data are calculated: `_coord_` means that vertex coordinates are compiled, `_vec_fct_` means that vector valued functions are calculated and so on.

Within these projection routines the vertex error estimators are used to decide whether a given datum has to be interpolated from parent data in order to assure continuity or if it has to be filled with the original value provided by the HMESH interface.

#### 11.3.4 The test-if-proceed routine

As mentioned above this function controls the traverse. It may return each of the three values `G_HM_NOACTION`, `G_HM_PROCEED` or `G_HM_ACTION`. If `G_HM_ACTION` is returned the traverse is stopped on the current element and the `action_on_element` routine is called. Whereas if `G_HM_NOACTION` is returned the traverse is stopped on the current element without calling the `action_on_element` routine. In contrast `G_HM_PROCEED` causes the traverse to go on to the next finer level. It is important to understand that the traverse mode determines on which elements the `action_on_element` routine may be called (on all nodes of the tree or only on the leaves) but the result of `test_if_proceed` determines when the `action_on_element` routine is actually called.

The arguments of the `test_if_proceed` routine are the `HELEMENT2D`, the `HM2_GENERAL`-structure and the element data of the current element. For an example see the `hm2_fct_lens_test_if_proceed` routine at the end of this section.

Typically the `test_if_proceed` routine compares an element error indicator to an appropriate threshold. In the 3D-case other intersection tests are also a common task performed in this routine. In the GRAPE-library several standard routines of this kind are incorporated. In the 2D-case these are:

- `hm2_geom_simpl_test_if_proceed`
- `hm2_geom_lens_test_if_proceed`
- `hm2_fct_simpl_test_if_proceed`
- `hm2_fct_lens_test_if_proceed`
- `hm2_geom_fct_simpl_test_if_proceed`

- hm2\_geom\_fct\_lens\_test\_if\_proceed

As the names indicate either the geometry error indicators, the function error indicators or both are compared to their respective threshold. Additionally the functionality of the lens is supported or not.

```
int hm2_fct_lens_test_if_proceed(HELEMENT2D* helement,
                                 HM2_GENERAL* general,
                                 void* el_data)
{
    HMESH2D*      hmesh      = (HMESH2D*)helement->mesh;
    F_HDATA2D*    f_hdata2d = general->f_hdata2d;
    HMESH2D_LENS* lens       = general->lens;

    double   error,lens_error;
    double   lens_threshold;

    /* get element error estimator */

    if(f_hdata2d->get_element_estimate)
        error = f_hdata2d->get_element_estimate(helement,
                                                f_hdata2d->function_data);
    else
        error = f_hdata2d->threshold + 1.0;

    /* lens condition */

    lens_threshold = lens->get_lens_element_threshold(helement,lens);

    if(hmesh->get_lens_element_estimate)
        lens_error = hmesh->get_lens_element_estimate(helement,
                                                       lens->lens_data);
    else
        lens_error = lens_threshold + 1.0;

    /* compare with threshold */

    if(error < f_hdata2d->threshold && lens_error < lens_threshold)
        return G_HM_ACTION;
    else
        return G_HM_PROCEED;
}
```

### 11.3.5 The action-on-element routine

This routine performs the actual rendering and displaying on a given element. The arguments are the HELEMENT2D, the HM2\_GENERAL-structure, the element data of the current element and a pointer to arbitrary, user defined data. The last argument is of the type `void*` and can be used to pass data from the display method to the `action_on_element` routine.

Below the `hm2_disp_draw_element` routine is printed as an example.

```
static int hm2_disp_draw_element(HELEMENT2D*  helement,
                                 HM2_GENERAL* general,
                                 void*         el_data,
                                 void*         action_arg)
{
    GRAPHICDEVICE*      dev      = general->dev;
```

```

HMESH2D*           hmesh = (HMESH2D*)helement->mesh;
HELEMENT2D_DESCRIPTION* descr =
    (HELEMENT2D_DESCRIPTION*)helement->descr;
HM2_COORD_DATA* element_data = (HM2_COORD_DATA*)el_data;

VEC3* vertex_coord = element_data->vertex_coord;
int i,number_of_vertices = descr->number_of_vertices;

if(dev->grid_patch == G_GRID){
    dev->move(vertex_coord[number_of_vertices-1]);
    for(i=0;i<number_of_vertices;i++)
        dev->draw(vertex_coord[i]);
}
else{
    VEC3 normal;

    g_vec3_get_normal_to_plane_quietly(normal,
                                         vertex_coord[0],
                                         vertex_coord[1],
                                         vertex_coord[2]);

    dev->begin_patch();
    dev->patch_normal(normal);

    for(i=0;i<number_of_vertices;i++)
        dev->patch_vertex(vertex_coord[i]);

    dev->end_patch();
}

return TRUE;
}

```

## 12 Hierarchical Searching using the Interface

Particle tracing methods which generate and visualize particle lines, stream surfaces or moving clouds of particles are important tools for the examination of velocity fields. The efficiency of these methods depends sensitively on searching algorithms. Let us suppose that  $h$  and  $\Delta t|u_{\max}|$  are of the same size in the corresponding integration method. Fig 22 sketches a typical particle line on a 2D grid. The basic task is to locate points on the grid, which are computed by the corresponding ODE solver. For each new position we need the grid element and the corresponding local coordinates to evaluate the numerical velocity at that position. It is obvious, that a local algorithm should be used in this searching process. This local method can be provided with an initial guess, which typically is the result of the last timestep. Figure 23 illustrates two alternative schemes. On the one hand, we can proceed recursively up and down. Thereby we first move from the guess positions fine grid element successively to coarser elements until the destination position is in the current element or we have reached the macro grid level. Then we zoom into finer grid level until the finest element which contains the destination position has been reached. On the other hand, we can start on the macro level, search for a coarse element containing the destination position and then perform the above zooming operation. Especially on hierarchical grids of a larger depth the first method is obviously the better choice. I. e. in average about four level changes (two up and two down respectively) are necessary for a 2D grid consisting of rectangles to locate the new point of the particle line under the above assumption on the step sizes. The kernel of the second algorithm could be implemented in the following way:

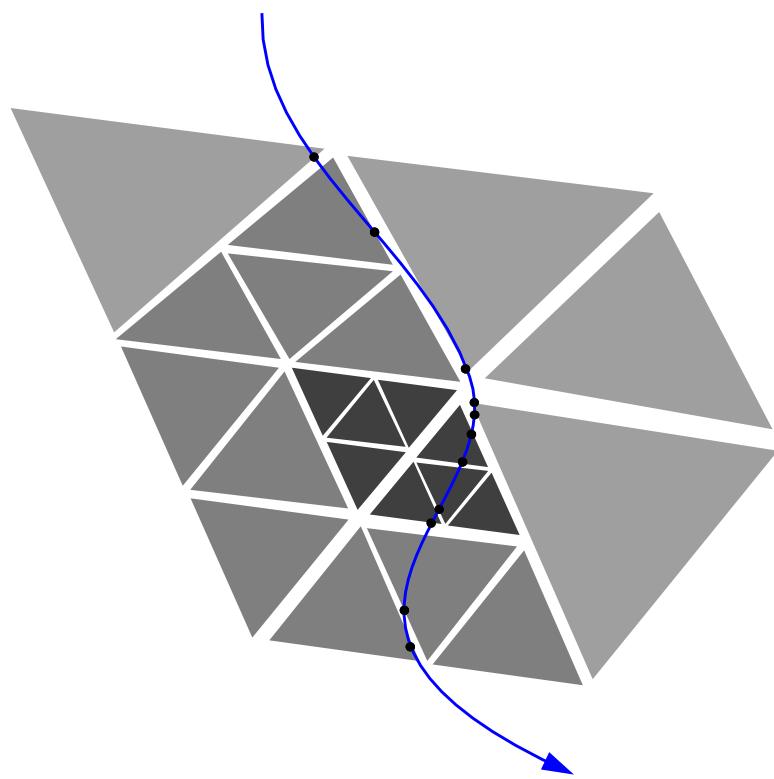


Figure 22: Typical particle line

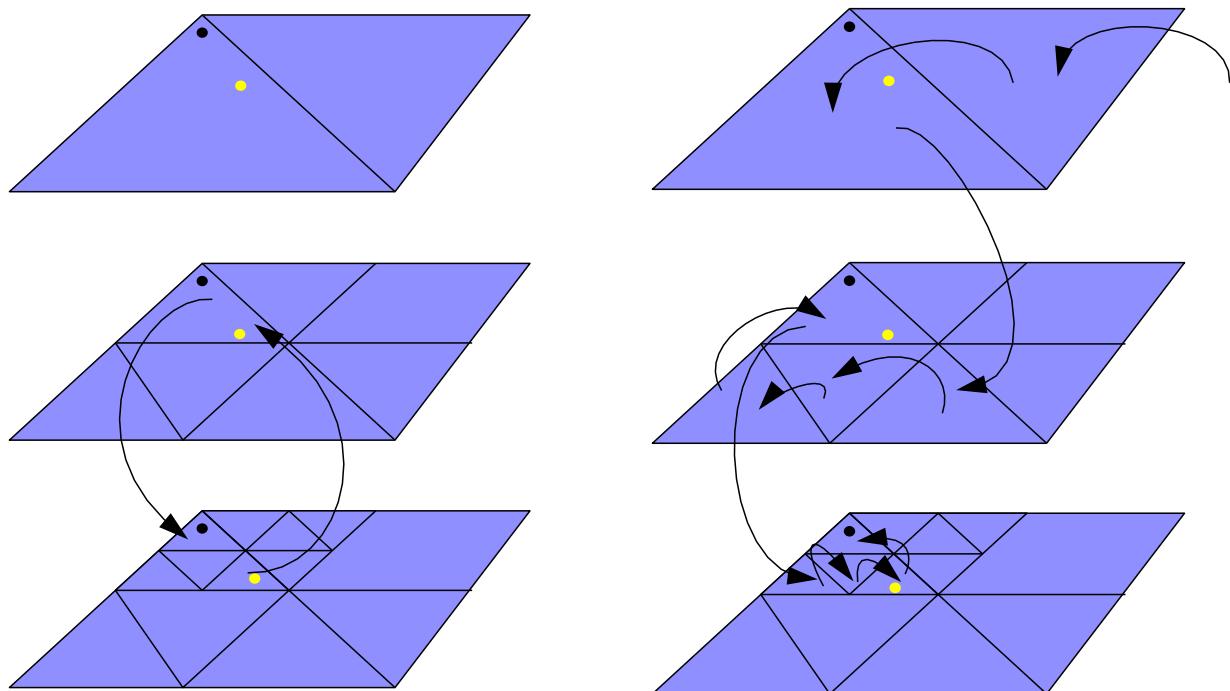


Figure 23: Different searching strategies

```

element = self->first_macro(hefAll);
element->descr->world_to_coord((ELEMENT*)element,point,local_coord);
while(element->descr->check_inside((ELEMENT*)element,local_coord)==false){
    element = self->next_macro(element,hefAll);
}

while((help_element = element->first_child(element,hefAll))!=NULL){
    element = help_element;

    element->descr->world_to_coord((ELEMENT*)element,point,local_coord);
    while(element->descr->check_inside((ELEMENT*)element,point)==false){
        element = element->next_child(element,hefAll);
    }
}

```

To improve efficiency the new interface routines

```
void coord_of_parent(HELEMENT3D *element, double *local_coord,
                     double *local_coord_of_parent);
```

and

```
HELEMENT3D *select_child(HELEMENT3D *element, double *local_coord,
                         double *local_coord_of_child,HELEMENT3D_FLAGS hefall)
```

have been added. *coord\_of\_parent()* transforms the local coordinates of a point in an element to that one of the parent element. *select\_child()* returns the child element containing a point already known to be inside the parent element and calculates its local coordinates corresponding to the child element. This allows very fast and direct access to elements of arbitrary nested hierarchical grids. With these functions at hand, the kernel for the first algorithm look as follows:

```

if(element != NULL){
    element->descr->world_to_coord((ELEMENT*)element,point,local_coord);
    while((-1 != element->desc->check_inside((ELEMENT3D *)element,local_coord))&&(e[0]->parent!=NULL))
        help_element = element;
        ((HELEMENT3D_DESCRIPTION*)(element->descr))->coord_of_parent
(element, local_coord,local_coord);
        element = element->parent;
        hmesh->free_element((ELEMENT3D*)help_element);
    }

    while((help_element = hmesh->select_child
(element,local_coord,local_coord,hefAll))!=NULL)
        {element = help_element;}
}

```

This algorithm is implemented in the method

```
HMESH3D *hmesh3d_search(VEC3 point, double *local_coord, HELEMENT3D **element)
```

which uses *\*element* as the element of the intial guess, to locate a particle at position *point* on the grid hierarchy.

## 13 An Interface Recipe

### 13.1 Introduction

In the following we will demonstrate which steps the user has to pass to get from a general interface mask to a complete hierarchical interface. In each step we will first present the

code from the interface mask (marked by horizontal lines) and afterwards discuss the necessary modifications. The recipe and the example cover the 2D-case. The 3D-case is analogous.

The interface consists of the following routines:

- HMESH2D-routines
  - get\_new\_hmesh()
  - Routines to traverse the hierarchical mesh
    - \* interface\_first\_macro()
    - \* interface\_next\_macro()
    - \* interface\_first\_child()
    - \* interface\_next\_child()
    - \* interface\_select\_child()
    - \* interface\_first\_element()
    - \* interface\_next\_element()
  - Handling interface\_elements
    - \* get\_interface\_element()
    - \* free\_interface\_element()
    - \* fill\_interface\_element\_with\_user\_macro\_element()
    - \* fill\_interface\_element\_with\_user\_child\_element()
  - Helement Description
    - \* element\_world\_to\_coord()
    - \* element\_coord\_to\_world()
    - \* helement\_description\_check\_inside()
    - \* helement\_description\_neighbour()
    - \* helement\_description\_boundary()
    - \* helement\_description\_coord\_of\_parent()
  - Geometry error estimator
    - \* get\_geometry\_vertex\_estimate()
    - \* get\_geometry\_element\_estimate()
- Routines for functions on hierachical meshes
  - get\_new\_fhdata()
  - data\_access\_function()
  - local\_information()
  - get\_bounds()
  - Function error estimator
    - \* get\_function\_vertex\_estimate()
    - \* get\_function\_element\_estimate()

In order to illustrate the single steps we introduce a sample data structure. As the interface uses its own data structures we have to distinguish three kinds of data structures:

- the data structure of the user
- the internal GRAPE data structures
- the interface data structures

In the example the user data structures are called `BISMESH`, `BELEMENTS` and `BELEMENT_MACRO`. The task of the interface is to translate these structures into the respective internal GRAPE structures `HMESSH2D` and `HELEMENT2D`. The interface employs so called `INTERFACE_ELEMENTS` to generate the `HELEMENT2Ds`.

### 13.2 The Bismesh Structure

Our sample data structure will describe a triangulation of a given domain that is refined by bisection starting from a macro triangulation. Each element is represented in the following data structure called `BELEMENT`:

```
typedef struct belement{
    struct belement* neighbour[3];
    struct belement* child[2];
    struct belement* parent;

    int newnode;
    int flag;
} BELEMENT;
```

Here `newnode` contains the global index of the node that is created while the element is being bisected. The `flag` is used to decide which element has to be refined. Elements of the macro triangulation called `BELEMENT_MACRO` are basically the same as the `BELEMENTs`. They only have some additional information included:

```
typedef struct belement_macro{
    BELEMENT* neighbour[3];
    BELEMENT* child[2];
    BELEMENT* parent;

    int newnode;
    int flag;

    int node[3];

    struct belement_macro* next;
} BELEMENT_MACRO;
```

The `BELEMENT_MACROs` are chain linked via the `next` pointer while the `node` array contains the global indices of the vertices.

Global information regarding the bisection mesh is grouped in the structure `BISMESH`:

```
typedef struct bismesh{
    int number_macro_points;
    int number_macro_elements;
    int number_points;
    int number_elements;
    int max_level;

    BELEMENT_MACRO* first;

    VEC3* macro_coord;

    double* geo_estimates;
} BISMESH;
```

The pointer `first` enables us to access the chain of the macro elements while the coordinates of the macro nodes are stored in the array `macro_coord`. The error estimators concerning the geometry are stored in the array `geo_estimates`.

### 13.3 Data Structures of the Interface

The purpose of a hmesh interface is to translate data stored in a given user format – like the above bismesh structures – into the HMESH2D/HELEMENT2D structures. According to the interface mask the INTERFACE\_ELEMENT consists of all data of the HELEMENT2D. These are grouped in HMESH2D\_HELEMENT2D\_STRUCT. The additional variables support the recursive mesh traversal and the generation of HELEMENT2Ds. Furthermore there may be some entries which depend on the user's data format.

---

```
typedef struct interface_element {
    HMESH2D_HELEMENT2D_STRUCT;

    USER_ELEMENT* user_element;
    double*      point_coords[MAX_NUMBER_OF_VERTICES_PER_ELEMENT];
    int          level;

    struct interface_element* next;
    ...
    /* additional entries by the user */

} INTERFACE_ELEMENT;
```

---

In our case the INTERFACE\_ELEMENT structure of the mask has the concrete outline:

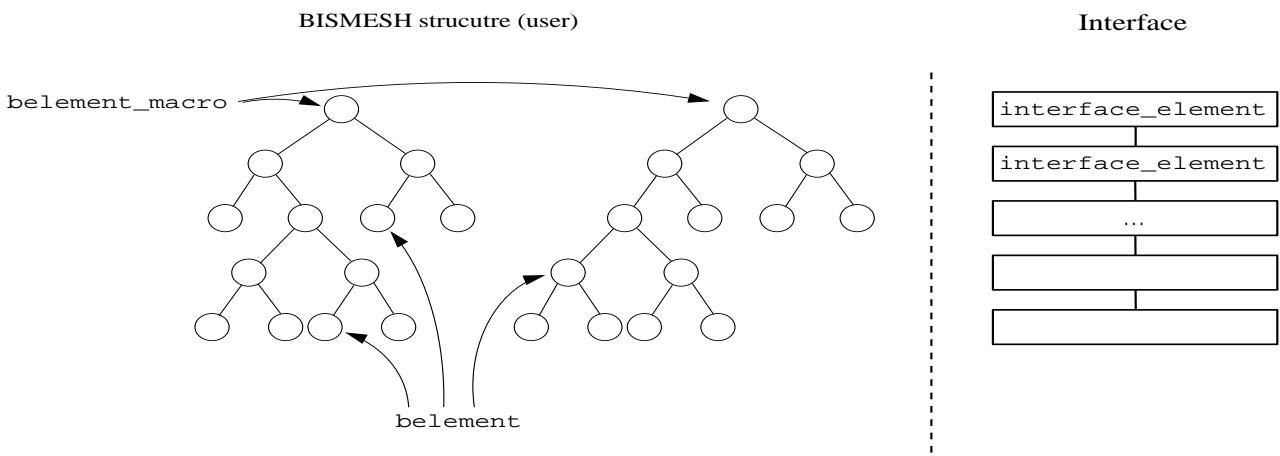
```
typedef struct interface_element {
    HMESH2D_HELEMENT2D_STRUCT;

    BELEMENT*      belement;
    double*        point_coords[MAX_NUMBER_OF_VERTICES_PER_ELEMENT];
    int            level;

    struct interface_element* next;
    VEC3           coord_of_new_node;

} INTERFACE_ELEMENT;
```

The INTERFACE\_ELEMENTS are organized in a free list which is linked via the `next` pointer. The coordinates of the point that is created while the element is being bisected are stored in `coord_of_new_node`. Although the routines for traversing the mesh have pointers to HELEMENT2Ds as arguments and return values, in the interface these pointers will be casted pointers to INTERFACE\_ELEMENTS. In an object oriented setting one would apply the subclass concept here.



### 13.4 get\_new\_hmesh()

The initialization routine of the interface mask gets a new instance of HMESH2D and fills in the neccessary data. In the interface mask it looks like that:

---

```

HMESH2D* get_new_hmesh(USER_MESH* user_mesh,...)
{
    HMESH2D* hmesh;

    hmesh = (HMESH2D*)GRAPE(HMesh2d,"new-instance")("name of hmesh");
    ASSURE(hmesh,"get_new_hmesh: can't get new hmesh instance",return NULL);

    hmesh->first_macro      = interface_first_macro;
    hmesh->next_macro       = interface_next_macro;
    hmesh->first_child       = interface_first_child;
    hmesh->next_child        = interface_next_child;
    hmesh->select_child      = interface_select_child;
    hmesh->first_element     = interface_first_element;
    hmesh->next_element      = interface_next_element;

    hmesh->max_level         = ...;
    hmesh->max_vindex         = ...;
    hmesh->level_of_interest  = ...;
    hmesh->dimension_of_world = ...;
    hmesh->max_dimension_of_coord = ...;
    hmesh->max_number_of_vertices = MAX_NUMBER_OF_VERTICES_PER_ELEMENT;
    hmesh->geometry_data      = (void*)...; /* 2D-case only */

    /* in 2D-case */

    hmesh->get_geometry_vertex_estimate =
        interface_get_geometry_vertex_estimate;

    hmesh->get_geometry_element_estimate =
        interface_get_geometry_element_estimate;

    /* fill the helement description */

    helement_description->number_of_vertices = ...;
    helement_description->dimension_of_coord = ...;
    helement_description->coord              = ...;
    helement_description->parametric_degree = ...;
    helement_description->world_to_coord    = ...;
    helement_description->coord_to_world    = ...;

    helement_description->check_inside     = helement_description_check_inside;
    helement_description->neighbour        = helement_description_neighbour;
    helement_description->boundary         = helement_description_boundary;
    helement_description->coord_of_parent  = helement_description_coord_of_parent;

    /* inheritance rules */

    inheritance_rule[0]->np      = ...; /* number of parent points */
    inheritance_rule[0]->pindex   = ...;
    inheritance_rule[0]->pweight  = ...;

    ...
    inheritance_rule[n]->np      = ...; /* number of parent points */
    inheritance_rule[n]->pindex   = ...;
    inheritance_rule[n]->pweight  = ...;

    return hmesh;
}

```

---

In our example this becomes:

```

HMESH2D* get_new_hmesh(BISMESH* bismesh)
{
    HMESH2D* hmesh;

    hmesh = (HMESH2D*)GRAPE(HMesh2d,"new-instance")("my sample bismesh");
    ASSURE(hmesh,"get_new_hmesh: can't get new hmesh instance",return NULL);

```

```

hmesh->first_macro          = interface_first_macro;
hmesh->next_macro           = interface_next_macro;
hmesh->first_child           = interface_first_child;
hmesh->next_child             = interface_next_child;
hmesh->select_child           = interface_select_child;
hmesh->first_element          = interface_first_element;
hmesh->next_element            = interface_next_element;

hmesh->max_level              = bismesh->max_level;
hmesh->max_vindex              = bismesh->number_points;
hmesh->user_data               = (void*)bismesh;
hmesh->level_of_interest        = 1; /* can be changed interactively later */
hmesh->dimension_of_world       = 3;
hmesh->max_dimension_of_coord = 3;
hmesh->max_number_of_vertices = MAX_NUMBER_OF_VERTICES_PER_ELEMENT;
hmesh->geometry_data           = NULL;

/* in 2D-case, see I.5 */

hmesh->get_geometry_vertex_estimate =
    get_geometry_vertex_estimate;

hmesh->get_geometry_element_estimate =
    get_geometry_element_estimate;

/* fill the helement description */

helement_description.number_of_vertices = 3;
helement_description.dimension_of_coord = 3; /* number of local coords
                                                 on an element */
helement_description.coord                = local_coordinate_system;
helement_description.parametric_degree   = 1;
helement_description.world_to_coord      = triangle_world_to_coord;
helement_description.coord_to_world       = triangle_coord_to_world;
helement_description.check_inside         = triangle_check_inside;

helement_description.neighbour           = helement_description_neighbour;
helement_description.boundary            = helement_description_boundary;
helement_description.coord_of_parent     = helement_description_coord_of_parent;

/* inheritance rules */

inheritance_rule_in_child_0[0] = vinherit_point_0;
inheritance_rule_in_child_0[1] = vinherit_point_1_in_child_0;
inheritance_rule_in_child_0[2] = vinherit_point_2_in_child_0;

inheritance_rule_in_child_1[0] = vinherit_point_0;
inheritance_rule_in_child_1[1] = vinherit_point_1_in_child_1;
inheritance_rule_in_child_1[2] = vinherit_point_2_in_child_1;

return hmesh;
}

```

The inheritance rules defined at the bottom describe how points in a child element can be expressed as a convex combination of points in the parent element.

For the bisection of triangles there are the following six inheritance rules:

```

static VINHERIT inheritance_rule_in_child_0[3];
static VINHERIT inheritance_rule_in_child_1[3];

static int      pindex_point_0[2] = {1 , 2 };
static double   pweight_point_0[2] = {0.5,0.5};
static VINHERIT vinherit_point_0   = {2,pindex_point_0,pweight_point_0};

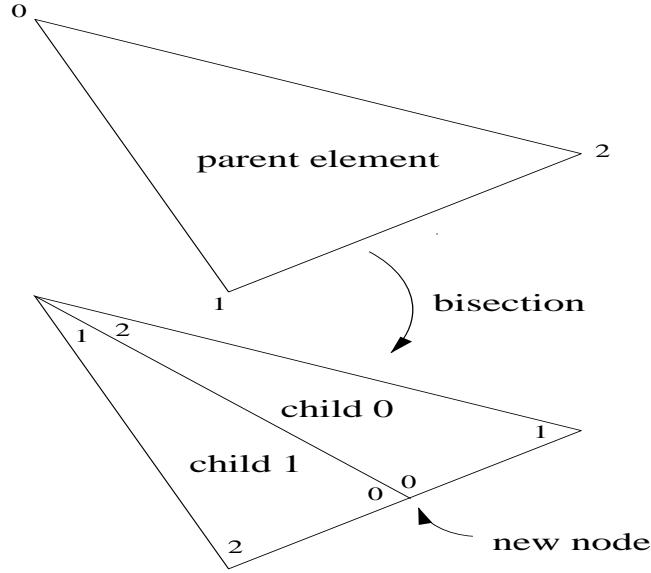
static double pweight_point_1_or_2[1] = {1.0};

static int      pindex_point_1_in_child_0[1] = {2};
static VINHERIT vinherit_point_1_in_child_0 =
    {1,pindex_point_1_in_child_0,pweight_point_1_or_2};

static int      pindex_point_2_in_child_0[1] = {0};
static VINHERIT vinherit_point_2_in_child_0 =
    {1,pindex_point_2_in_child_0,pweight_point_1_or_2};

static int      pindex_point_1_in_child_1[1] = {0};
static VINHERIT vinherit_point_1_in_child_1 =
    {1,pindex_point_1_in_child_1,pweight_point_1_or_2};

```



```
static int      pindex_point_2_in_child_1[1] = {1};  
static VINHERIT vinherit_point_2_in_child_1 =  
    {1,pindex_point_2_in_child_1,pweight_point_1_or_2};
```

The meaning of the VINHERIT are discussed in detail in 10.

### 13.5 Routines to Traverse the hmesh

The following routines are used by GRAPE to traverse the mesh. The main modification from the code in the interface mask is to provide the respective element of the user data structure, i.e. in our case the `BELEMENT` or `BELEMENT_MACRO`.

**interface\_first\_macro()**

This routine gets the first macro element of the user mesh. In the interface mask it looks like that:

```

static HELEMENT2D* interface_first_macro(HMESH2D* hmesh,
                                         MESH_ELEMENT_FLAGS flags)
{
    USER_MACRO_ELEMENT* user_macro      = ...;
    INTERFACE_ELEMENT*  interface_element = get_interface_element(hmesh);

    fill_interface_element_with_user_macro_element(interface_element,
                                                    user_macro,
                                                    flags);

    return (HELEMENT2D*)interface_element;
}

```

In our bismesh structure we have a pointer to the first macro element at hand. We use the interface routines `get_interface_element()` to request an empty `INTERFACE_ELEMENT` from the free list and `fill_interface_element_with_user_macro_element()` to write the data of `belement_macro` in the `INTERFACE_ELEMENT`.

```

BISMESH*           bismesh          = (BISMESH*)(hmesh->user_data);
BELEMENT_MACRO*   belement_macro   = bismesh->first;
INTERFACE_ELEMENT* interface_element = get_interface_element(hmesh);

fill_interface_element_with_user_macro_element(interface_element,
                                               belement_macro,
                                               flags);

return (HELEMENT2D*)interface_element;
}

```

**interface\_next\_macro()**

This routine gets the next macro element of a given macro element. If there is no next macro element the used INTERFACE\_ELEMENT is returned to the free list.

---

```

static HELEMENT2D* interface_next_macro(HELEMENT2D*      helement,
                                         MESH_ELEMENT_FLAGS flags)
{
    INTERFACE_ELEMENT* interface_element = (INTERFACE_ELEMENT*)helement;
    USER_MACRO_ELEMENT* user_macro     = ...;

    if(user_macro){
        fill_interface_element_with_user_macro_element(interface_element,
                                                       user_macro,
                                                       flags);
    }
    else{
        free_interface_element(interface_element);
        interface_element = NULL;
    }

    return (HELEMENT2D*)interface_element;
}

```

---

This translates into:

```

static HELEMENT2D* interface_next_macro(HELEMENT2D*      helement,
                                         MESH_ELEMENT_FLAGS flags)
{
    INTERFACE_ELEMENT* interface_element = (INTERFACE_ELEMENT*)helement;
    BELEMENT_MACRO*   belement_macro   =
        ((BELEMENT_MACRO*)interface_element->belement)->next;

    if(belement_macro){
        fill_interface_element_with_user_macro_element(interface_element,
                                                       belement_macro,
                                                       flags);
    }
    else{
        free_interface_element(interface_element);
        interface_element = NULL;
    }

    return (HELEMENT2D*)interface_element;
}

```

**interface\_first\_child()**

This routine gets the first child element of a given element.

---

```

static HELEMENT2D* interface_first_child(HELEMENT2D*      helement,
                                         MESH_ELEMENT_FLAGS flags)
{
    INTERFACE_ELEMENT* interface_child_element;
    INTERFACE_ELEMENT* interface_parent_element = (INTERFACE_ELEMENT*)helement;

    USER_ELEMENT* user_element = ...;
    int level_of_interest = ((HMESH2D*)helement->mesh)->level_of_interest;

```

```

if(user_element && (interface_parent_element->level < level_of_interest)){
    interface_child_element = get_interface_element((HMESH2D*)helement->mesh);
    fill_interface_element_with_user_child_element(interface_child_element,
                                                   interface_parent_element,
                                                   user_element,
                                                   flags);
    return (HELEMENT2D*)interface_child_element;
}
else
    return NULL;
}

```

---

In our example this reads as follows:

```

static HELEMENT2D* interface_first_child(HELEMENT2D*           helement,
                                         MESH_ELEMENT_FLAGS flags)
{
    INTERFACE_ELEMENT* interface_child_element;
    INTERFACE_ELEMENT* interface_parent_element = (INTERFACE_ELEMENT*)helement;

    BELEMENT* belement = interface_parent_element->belement->child[0];
    int level_of_interest = ((HMESH2D*)helement->mesh)->level_of_interest;

    if(belement && (interface_parent_element->level < level_of_interest)){
        interface_child_element = get_interface_element((HMESH2D*)helement->mesh);
        fill_interface_element_with_user_child_element(interface_child_element,
                                                       interface_parent_element,
                                                       belement,
                                                       flags);
        return (HELEMENT2D*)interface_child_element;
    }
    else
        return NULL;
}

```

### **interface\_next\_child()**

This routine returns the next child element of the parent of the given (child) element. If there is no next child element the used INTERFACE\_ELEMENT is returned to the free list.

```

static HELEMENT2D* interface_next_child(HELEMENT2D*           helement,
                                         MESH_ELEMENT_FLAGS flags)
{
    INTERFACE_ELEMENT* interface_child_element = (INTERFACE_ELEMENT*)helement;
    INTERFACE_ELEMENT* interface_parent_element =
        (INTERFACE_ELEMENT*)helement->parent;

    USER_ELEMENT* user_element = ...;

    if(user_element){
        fill_interface_element_with_user_child_element(interface_child_element,
                                                       interface_parent_element,
                                                       user_element,
                                                       flags);
        return (HELEMENT2D*)interface_child_element;
    }
    else{ /* there exists no further child\_\_element */
        free_interface_element(interface_child_element);
        return NULL;
    }
}

```

---

To adapt this to our BISMESH case we have to modify it as follows:

```

static HELEMENT2D* interface_next_child(HELEMENT2D*           helement,
                                         MESH_ELEMENT_FLAGS flags)
{
    INTERFACE_ELEMENT* interface_child_element = (INTERFACE_ELEMENT*)helement;
    INTERFACE_ELEMENT* interface_parent_element =
        (INTERFACE_ELEMENT*)helement->parent;

```

```

if(interface_child_element->belement ==
   interface_parent_element->belement->child[0]){
    fill_interface_element_with_user_child_element(interface_child_element,
                                                   interface_parent_element,
                                                   interface_parent_element->belement->child[1],
                                                   flags);

    return (HELEMENT2D*)interface_child_element;
}
else{
    free_interface_element(interface_child_element);
    return NULL;
}
}

```

### interface\_select\_child()

The purpose of this routine is explained in 10. In our sample case we have:

```

static HELEMENT2D* interface_select_child(HELEMENT2D* parent_helement,
                                         double*      local_parent_coord,
                                         double*      local_child_coord,
                                         MESH_ELEMENT_FLAGS flags)
{
    HELEMENT2D*      child_helement;
    INTERFACE_ELEMENT* interface_child_element;
    BELEMENT*        belement;

    if(local_parent_coord[1] < local_parent_coord[2]){
        belement = ((INTERFACE_ELEMENT*)parent_helement)->belement->child[0];
        if(belement){
            local_child_coord[0] = local_parent_coord[1] * 2.0;
            local_child_coord[2] = local_parent_coord[0];
            local_child_coord[1] = 1.0 - local_child_coord[0] - local_child_coord[2];
        }
    }
    else{
        belement = ((INTERFACE_ELEMENT*)parent_helement)->belement->child[1];
        if(belement){
            local_child_coord[0] = local_parent_coord[2] * 2.0;
            local_child_coord[1] = local_parent_coord[0];
            local_child_coord[2] = 1.0 - local_child_coord[0] - local_child_coord[1];
        }
    }

    if(belement){
        interface_child_element =
            get_interface_element((HMESH2D*)parent_helement->mesh);

        fill_interface_element_with_user_child_element(interface_child_element,
                                                       (INTERFACE_ELEMENT*)parent_helement,
                                                       belement,
                                                       flags);

        return (HELEMENT2D*)interface_child_element;
    }
    else{
        return NULL;
    }
}

```

### interface\_first\_element()

This is a standard-mesh2d-routine and returns the first element on the deepest level described by hmesh->level\_of\_interest. It is independent of the user's data structures and does not need to be changed in any way.

---

```

static ELEMENT2D* interface_first_element(MESH2D* mesh,
                                         MESH_ELEMENT_FLAGS flags)
{
    HMESH2D*      hmesh      = (HMESH2D*)mesh;
    HELEMENT2D*    helement  = hmesh->first_macro(hmesh,flags);

```

---

```

HELEMENT2D* auxiliary_helement;
INTERFACE_ELEMENT* interface_element = (INTERFACE_ELEMENT*)helement;

while(auxiliary_helement = hmesh->first_child(helement,flags)){
    helement      = auxiliary_helement;
    interface_element = (INTERFACE_ELEMENT*)helement;
}

return (ELEMENT2D*)helement;
}

```

---

**interface\_next\_element()**

Like `interface_first_element()` this is a standard-mesh2d-routine and it returns the next element on the deepest level described by `hmesh->level_of_interest`. It is independent of the user's data structures, too. Therefore it does not need to be changed either.

---

```

static ELEMENT2D* interface_next_element(ELEMENT2D*           element,
                                         MESH_ELEMENT_FLAGS flags)
{
    HMESH2D*     hmesh      = (HMESH2D*)helement->mesh;
    HELEMENT2D*   helement   = (HELEMENT2D*)element;
    HELEMENT2D*   helement_next;

    if(helement->parent){
        helement_next = hmesh->next_child(helement,flags);
        if(!helement_next)
            helement_next = (HELEMENT2D*)interface_next_element((ELEMENT2D*)
                                         helement->parent,flags);
    }
    else
        helement_next = hmesh->next_macro(helement,flags);

    if(helement_next){
        while(helement = hmesh->first_child(helement_next,flags))
            helement_next = helement;
    }

    return (ELEMENT2D*)helement_next;
}

```

---

**13.6 Handling INTERFACE\_ELEMENTs**

Now we list and describe fundamental procedures which support the handling of elements in the interface.

**get\_interface\_element()**

This routine returns a new `INTERFACE_ELEMENT`. Either there is an empty one in the free list or a completely new one is generated. As the additional data in the `INTERFACE_ELEMENT` in our example does not need any special memory treatment there are no modifications necessary.

---

```

static INTERFACE_ELEMENT* get_interface_element(HMESH2D* hmesh)
{
    INTERFACE_ELEMENT* interface_element;

    if(interface_free_list_first_entry){
        /*
         * We have an interface_element in our free list. It was generated
         * somewhen in the past and freed afterwards. So we can use the
         * allocated memory again.
        */
    }

```

```

*/
interface_element           = interface_free_list_first_entry;
interface_free_list_first_entry = interface_element->next;
}
else {
/*
    There is no interface_element in our free list, so we have to
    generate a new one.
*/
interface_element =
    (INTERFACE_ELEMENT*)mem_alloc(sizeof(INTERFACE_ELEMENT));

interface_element->next   = NULL;

interface_element->mesh   = (MESH2D*)hmesh;
interface_element->vertex = (double**)interface_element->point_coords;
interface_element->vindex = int_alloc(MAX_NUMBER_OF_VERTICES_PER_ELEMENT);
interface_element->vih    = mem_alloc(MAX_NUMBER_OF_VERTICES_PER_ELEMENT *
                                      sizeof(VINHERIT));
}

return interface_element;
}

```

---

**free\_interface\_element()**

If data stored in `interface_element` are no longer used, the `INTERFACE_ELEMENT` can be appended to the free list. The routine does not need to be changed.

```

static void free_interface_element(INTERFACE_ELEMENT* interface_element)
{
    interface_element->next           = interface_free_list_first_entry;
    interface_free_list_first_entry = interface_element;

    return;
}

```

---

Let us emphasize that all free list entries contain useful initialized values which will be reused in future requests.

**fill\_interface\_element\_with\_user\_macro\_element()**

This routine fills an `INTERFACE_ELEMENT` with the data of a user macro element.

```

static void fill_interface_element_with_user_macro_element(
    INTERFACE_ELEMENT* interface_element,
    USER_MACRO_ELEMENT* user_macro,
    MESH_ELEMENT_FLAGS flags)
{
    int number_of_vertices = ...;
    int i;

    /* element2d-entries */

    if(flags & hefVindex)
        for(i=0;i<number_of_vertices;i++)
            interface_element->vindex[i] = ...;

    interface_element->eindex    = ...;
    interface_element->descr     = (ELEMENT2D_DESCRIPTION*)...
    interface_element->present   = flags;
    interface_element->user_data = NULL;

    /* helement2d-entries */
}

```

---

```

interface_element->parent = NULL;
interface_element->ref_rule = ...;

/* interface_element-entries */

if(flags & hefVertex)
    for(i=0;i<number_of_vertices;i++)
        interface_element->point_coords[i] = ...;

interface_element->user_element = (USER_MACRO_ELEMENT*)user_macro;
interface_element->level = 0;

return;
}

```

---

The BISMESH version of it looks like:

```

static void fill_interface_element_with_user_macro_element(
    INTERFACE_ELEMENT* interface_element,
    BELEMENT_MACRO* belement_macro,
    MESH_ELEMENT_FLAGS flags)
{
    HMESH2D* hmesh           = (HMESH2D*)interface_element->mesh;
    BISMESH* bismesh         = (BISMESH*)hmesh->user_data;
    double norm              = 0.0;
    int number_of_vertices   = 3;
    int i;

    /* element2d-entries */

    for(i=0;i<number_of_vertices;i++)
        interface_element->vindex[i] = belement_macro->node[i];

    interface_element->descr      = (ELEMENT2D_DESCRIPTION*)&helement_description;
    interface_element->present    = flags;
    interface_element->user_data = NULL;

    /* helement2d-entries */

    interface_element->parent = NULL;

    /* interface_element-entries */

    for(i=0;i<number_of_vertices;i++)
        interface_element->point_coords[i] =
            bismesh->macro_coord[belement_macro->node[i]];

    interface_element->belement = (BELEMENT*)belement_macro;
    interface_element->level = 0;

    for(i=0;i<3;i++){
        interface_element->coord_of_new_node[i] =
            0.5 * interface_element->point_coords[1][i] +
            0.5 * interface_element->point_coords[2][i];

        norm += interface_element->coord_of_new_node[i] *
            interface_element->coord_of_new_node[i];
    }

    norm = g_sqrt(norm) / 0.5;

    for(i=0;i<3;i++)
        interface_element->coord_of_new_node[i] /= norm;

    return;
}

```

---

At the bottom of the routine the coordinates of the new node are calculated. Here we have to take the geometry of the mesh into account. In our example the mesh geometry is simply a sphere with the radius 0.5.

---

**fill\_interface\_element\_with\_user\_child\_element()**

This routine serves the purpose of filling an INTERFACE\_ELEMENT with a non macro element. The necessary data either comes from the respective user element or is derived from the parent INTERFACE\_ELEMENT.

---

```
static void fill_interface_element_with_user_child_element(
    INTERFACE_ELEMENT* interface_child_element,
    INTERFACE_ELEMENT* interface_parent_element,
    USER_ELEMENT*      user_element,
    MESH_ELEMENT_FLAGS flags)
{
    int number_of_vertices = ...;
    int i;

    /* element2d-entries */

    if(flags & hefVindex)
        for(i=0;i<number_of_vertices;i++)
            interface_element->vindex[i] = ...;

    interface_element->eindex    = ...;
    interface_element->descr     = (ELEMENT2D_DESCRIPTION*)...;
    interface_element->present   = flags;
    interface_element->user_data = NULL;

    /* helement2d-entries */

    interface_element->parent = (HELEMENT2D*)interface_parent_element;
    if(flags & hefVinh)
        for(i=0;i<number_of_vertices;i++)
            interface_element->vindh[i] = ...;

    interface_element->ref_rule = ...;

    /* interface_element-entries */

    if(flags & hefVertex)
        for(i=0;i<number_of_vertices;i++)
            interface_element->point_coords[i] = ...;

    interface_element->user_element = user_element;
    interface_element->level = ...;

    return;
}
```

---

The necessary modifications in our example are listed below:

```
static void fill_interface_element_with_user_child_element(
    INTERFACE_ELEMENT* interface_child_element,
    INTERFACE_ELEMENT* interface_parent_element,
    BELEMENT*          belement,
    MESH_ELEMENT_FLAGS flags)
{
    double norm = 0.0;
    int number_of_vertices = 3;
    int i;

    /* element2d-entries */

    interface_child_element->vindex[0] =
        interface_parent_element->belement->newnode;

    if(belement == interface_parent_element->belement->child[0]){
        interface_child_element->vindex[1] =
            interface_parent_element->vindex[2];
        interface_child_element->vindex[2] =
            interface_parent_element->vindex[0];
    }
    else{
        interface_child_element->vindex[1] =
            interface_parent_element->vindex[0];
        interface_child_element->vindex[2] =
            interface_parent_element->vindex[1];
    }
}
```

```

interface_child_element->descr =
  (ELEMENT2D_DESCRIPTION*)&helement_description;
interface_child_element->present = flags;
interface_child_element->user_data = NULL;

/* helement2d-entries */

interface_child_element->parent = (HELEMENT2D*)interface_parent_element;
if(flags & hefVinh){
  if(belement == interface_parent_element->belement->child[0]){
    for(i=0;i<number_of_vertices;i++)
      interface_child_element->vinh[i] = inheritance_rule_in_child_0[i];
  }
  else{
    for(i=0;i<number_of_vertices;i++)
      interface_child_element->vinh[i] = inheritance_rule_in_child_1[i];
  }
}

/* interface_element-entries */

if(belement == interface_parent_element->belement->child[0]){
  interface_child_element->point_coords[0] =
    interface_parent_element->coord_of_new_node;
  interface_child_element->point_coords[1] =
    interface_parent_element->point_coords[2];
  interface_child_element->point_coords[2] =
    interface_parent_element->point_coords[0];
}
else{
  interface_child_element->point_coords[0] =
    interface_parent_element->coord_of_new_node;
  interface_child_element->point_coords[1] =
    interface_parent_element->point_coords[0];
  interface_child_element->point_coords[2] =
    interface_parent_element->point_coords[1];
}

for(i=0;i<3;i++){
  interface_child_element->coord_of_new_node[i] =
    0.5 * interface_child_element->point_coords[1][i] +
    0.5 * interface_child_element->point_coords[2][i];

  norm += interface_child_element->coord_of_new_node[i] *
    interface_child_element->coord_of_new_node[i];
}
norm = g_sqrt(norm) / 0.5;

for(i=0;i<3;i++)
  interface_child_element->coord_of_new_node[i] /= norm;

interface_child_element->belement = belement;
interface_child_element->level = interface_parent_element->level + 1;

return;
}

```

Here like in `fill_interface_element_with_user_macro_element()` we have chosen a sphere geometry.

### 13.7 Helement Description

Remark: If you are only interested in the use of the standard GRAPE display methods on HMESHes (as to depict function values, extract isolines resp. isosurfaces, clip objects) you don't have to support the routines in the structure HELEMENT\_DESCRIPTION. (for these methods the routines are not used at all)

#### `element_world_to_coord()`

For given world coordinates  $x = \text{world}[0]$ ,  $y = \text{world}[1]$  and  $z = \text{world}[2]$  the routine calculates the coordinates in the local coordinate system of the element. The result is written into

`coord[]`. The return value is identical to that from `check_inside()`.

---

```
static int element_world_to_coord(HELEMENT2D* helement,
                                 double*      world,
                                 double*      coord)
{
    coord[0] = ...;
    coord[1] = ...;
    coord[2] = ...;

    return(helement->descr->check_inside(helement,coord));
}
```

---

In our example we have only triangles as elements and this routine reads:

```
static int element_world_to_coord(HELEMENT2D* helement,
                                 double*      world,
                                 double*      coord)
{
    double a[2][2];
    VEC2 b,c;
    int i,j;

    ASSURE(helement,
           "hmesh2d: triangle_world_to_coord: no element!",
           return(0));

    ASSURE(helement->vertex,
           "hmesh2d: triangle_world_to_coord: no coordinates!",
           return(0));

    for(i=0;i<2;i++)
        for(j=0;j<2;j++)
            a[j][i] = helement->vertex[i][j] - helement->vertex[2][j];

    for(j=0;j<2;j++)
        b[j] = world[j] - helement->vertex[2][j];

    if(!g_solve2(a,b,c)){
        fprintf(stderr,"hmesh2d-world-to-coord: point not in helement!\n");
        return(-2);
    }

    coord[0] = c[0];
    coord[1] = c[1];
    coord[2] = 1.0 - c[0] - c[1] ;

    return(helement->descr->check_inside(helement,coord));
}
```

### `element_coord_to_world()`

For given coordinates in the local coordinate system  $x = \text{coord}[0]$ ,  $y = \text{coord}[1]$  and  $z = \text{coord}[2]$  of an element the routine calculates the coordinates in world coordinates. The result is written into `world[]`.

---

```
static void element_coord_to_world(HELEMENT2D* helement,
                                  double*      coord,
                                  double*      world)
{
    world[0] = ...;
    world[1] = ...;
    world[2] = ...;

    return;
}
```

---

For our case that translates to:

```

static void element_coord_to_world(HELEMENT2D* helement,
                                  double*      coord,
                                  double*      world)
{
    int i,j;

    ASSURE(helement,
           "hmesh2d: triangle_coord_to_world: no element!",
           return);

    ASSURE(helement->vertex,
           "hmesh2d: triangle_coord_to_world: no coordinates!",
           return);

    for(i=0;i<3;i++){
        world[i] = coord[0] * helement->vertex[0][i];

        for(j=1;j<3;j++)
            world[i] += coord[j] * helement->vertex[j][i];
    }

    return;
}

```

### helement\_description\_check\_inside()

returns INSIDE (-1), if the point with local coordinates `coord[]` is inside of the element's parameter range. Otherwise, at least one plane of a local face separates the element and the given point. In this case the local index of such a face is returned.

---

```

static int helement_description_check_inside(HELEMENT2D* helement,
                                             double*      coord)
{
    return_index = ...;

    return(return_index);
}

```

---

We could write it as:

```

static int helement_description_check_inside(HELEMENT2D* helement,
                                             double*      coord)
{
    double minimum      = -1.0E-5;
    int   return_index = -1;
    int   i;

    ASSURE(helement,
           "hmesh2d: helement_description_check_inside: no element!",
           return);

    for(i=0;i<3;i++){
        if(coord[i] < minimum){
            minimum      = coord[i];
            return_index = i;
        }
    }

    return(return_index);
}

```

### helement\_description\_neighbour()

Returns a pointer to the adjacent element of the element across the edge with the given number at a point with local coordinates `coord`. After the call, `coord` contains the coordinates of the point transformed to the neighbouring element's local coordinate system. The structure `helement2d` may be overwritten.

`coord = NULL` is possible, then one of the neighbouring elements across the face will be returned.

flag :

EXACT\_NEIGHBOUR -1 (coord has to be on the element boundary, then the neighbouring element with the coord point as a boundary point is returned)  
 FIRST\_NEIGHBOUR 0 (return the first neighbour on this face)  
 NEXT\_NEIGHBOUR 1 (succesively returns the list of neighbours)

---

```
static HELEMENT2D* helement_description_neighbour(HELEMENT2D* helement,
                                                int edge,
                                                int flag,
                                                double* coord,
                                                double* world,
                                                MESH_ELEMENT_FLAGS flags)
{
    INTERFACE_ELEMENT* interface_element;
    USER_ELEMENT* user_element;
    USER_ELEMENT* neighbour_element = NULL;

    ASSURE(helement,
           "hmesh2d: helement_description_neighbour: no element!",
           return(NULL));

    interface_element = (INTERFACE_ELEMENT*)helement;
    user_element = interface_element->user_element;

    ...; /* get the appropriate neighbour */
    neighbour_element = ...; /* element, if possible. */

    if(neighbour_element){
        ...; /* fill interface_element with the */
        ...; /* necessary data */
    }

    /* remark: in the hierarchical setting some data on an element are obtained
       from the data on its parent element by inheritance;
       so to fill the interface_element with the appropriate
       neighbour_element data you have to supply the interface_element
       belonging to neighbour_element->parent too.
    */
}
else{
    ...; /* free the element */
}

return(helement);
}
```

---

### helement\_description\_boundary()

Returns 0, if edge is an inner edge of the mesh, or a nonzero integer, if edge is part of the mesh's boundary.

---

```
static int helement_description_boundary(HELEMENT2D* helement,int edge)
{
    INTERFACE_ELEMENT* interface_element;
    USER_ELEMENT* user_element;

    ASSURE(helement,
           "hmesh2d: helement_description_boundary: no element!",
           return(NULL));

    interface_element = (INTERFACE_ELEMENT*)helement;
    user_element = interface_element->user_element;

    ...; /* see if there is a neighbour across */
    /* the edge */
    if(/* neighbour exists */)
        return 0;
    else
        return 1;
}
```

We modify this into:

```

static int helement_description_boundary(HELEMENT2D* helement,int edge)
{
    ASSURE(helement,
           "hmesh2d: helement_description_boundary: no element!",
           return(NULL));

    if(helement_description_neighbour(helement,edge,NULL,NULL,NULL))
        return 0;
    else
        return 1;
}

```

**helement\_description\_coord\_of\_parent()**

Given a point in an element by its local coordinates this routine calculates the local coordinates of this point in the parent element

```

static void helement_description_coord_of_parent(HELEMENT2D* helement,
                                                double*      coord,
                                                double*      parent_coord);
{
    int i;

    ASSURE(helement,
           "hmesh2d: helement_description_coord_of_parent: no element!",
           return(NULL));

    ...

    for(i=0;i<3;i++)
        parent_coord[i] = ...;

    return;
}

```

For our BISMESH consisting of triangles we can formulate this like:

```

static void helement_description_coord_of_parent(HELEMENT2D* helement,
                                                double*      coord,
                                                double*      parent_coord);
{
    INTERFACE_ELEMENT* interface_element = (INTERFACE_ELEMENT*)helement;
    BELEMENT*          belement        = interface_element->belement;

    parent_coord[0] = coord[0];
    if(belement->node[1] == belement->parent->newnode){
        parent_coord[1] = .5 * coord[1];
        parent_coord[2] = 1 - parent_coord[0] - parent_coord[1];
    }
    else{
        parent_coord[2] = .5 * coord[2];
        parent_coord[1] = 1 - parent_coord[0] - parent_coord[2];
    }
    return;
}

```

### 13.8 Geometry Error Estimator

**get\_geometry\_vertex\_estimate()**

This routine fills the array `results` with the error estimators of the vertices of the element.

---

```

int number_of_vertices = ...;
int i;

for(i=0;i<number_of_vertices;i++)
    results[i] = ...;

return;
}

```

---

In our example we have a point based error estimator, which is calculated in a pre-roll step and stored in the array `geometry_data`.

```

static void get_geometry_vertex_estimate(HELEMENT2D* helement,
                                         double*      results,
                                         void*        geometry_data)
{
    int number_of_vertices = 3;
    int i;

    for(i=0;i<number_of_vertices;i++)
        results[i] = ((double*)geometry_data)[helement->vindex[i]];

    return;
}

```

#### `get_geometry_element_estimate()`

This routine returns the error estimator of the element.

---

```

static double get_geometry_element_estimate(HELEMENT2D* helement,
                                           void*        geometry_data)
{
    return ...;
}

```

---

The element estimator we choose is defined as the vertex estimator of the new node.

```

static double get_geometry_element_estimate(HELEMENT2D* helement,
                                           void*        geometry_data)
{
    BELEMENT* belement = ((INTERFACE_ELEMENT*)helement)->belement;

    return ((double*)geometry_data)[belement->newnode];
}

```

If different refinement rules with more new nodes on the next grid level are applied all these nodal error indicators have to be taken into account.

### 13.9 Routines for Functions on hmeshes

#### `get_new_fhdata()`

In analogy to `get_new_hmesh()` this routine generates a `f_hdata2d` structure. GRAPE uses `f_hdata2d` structures to access data on a function on a `hmesh2d`-object. In the interface mask this looks like:

---

```

F_HDATA2D* get_new_fhdata(...)
{
    F_HDATA2D* f_hdata2d = (F_HDATA2D*)mem_alloc(sizeof(F_HDATA2D));

    f_hdata2d->name          = "name of function";
    f_hdata2d->dimension_of_value = ...;
    f_hdata2d->continuous_data = ...; /* 0 or 1 */
    f_hdata2d->f              = data_access_function;
    f_hdata2d->f_el_info      = local_information;
}

```

```

f_hdata2d->user_data          = ...;
f_hdata2d->last               = ...; /* if there are other */
f_hdata2d->next               = ...; /* functions           */
f_hdata2d->function_data      = ...;
f_hdata2d->get_bounds          = get_bounds;
f_hdata2d->get_vertex_estimate = get_function_vertex_estimate;
f_hdata2d->get_element_estimate = get_function_element_estimate;
f_hdata2d->threshold          = 0.0;
f_hdata2d->hp_threshold        = 0.0;
f_hdata2d->hp_maxlevel         = 0;

return f_hdata2d;
}

```

---

In the sample program it has been adapted to:

```

F_HDATA2D* get_new_fhdata(double* function_values,double* error_estimates)
{
    F_HDATA2D* f_hdata2d = (F_HDATA2D*)mem_alloc(sizeof(F_HDATA2D));
    USER_FUNCTION_DATA* user_function_data;

    user_function_data =
        (USER_FUNCTION_DATA*)mem_alloc(sizeof(USER_FUNCTION_DATA));

    user_function_data->function_values = function_values;
    user_function_data->error_estimates = error_estimates;

    f_hdata2d->name          = "name of function";
    f_hdata2d->dimension_of_value = 1;
    f_hdata2d->continuous_data = 1; /* 0 or 1 */
    f_hdata2d->f              = data_access_function;
    f_hdata2d->f_el_info      = local_information;
    f_hdata2d->user_data       = NULL;
    f_hdata2d->last            = NULL; /* if there are other */
    f_hdata2d->next            = NULL; /* functions           */
    f_hdata2d->function_data   = (void*)user_function_data;
    f_hdata2d->get_bounds       = get_bounds;
    f_hdata2d->get_vertex_estimate = get_function_vertex_estimate;
    f_hdata2d->get_element_estimate = get_function_element_estimate;
    f_hdata2d->threshold        = 0.0;
    f_hdata2d->hp_threshold     = 0.0;
    f_hdata2d->hp_maxlevel      = 0;

    return f_hdata2d;
}

```

Here `USER_FUNCTION_DATA` is defined as:

```

typedef struct user_function_data {
    double* function_values;
    double* error_estimates;
} USER_FUNCTION_DATA;

```

After creating the `f_hdata2d` structure, you have to fill the hmesh with:

```
hmesh->f_data = (GENMESH_FDATA*)f_hdata2d;
```

If you want to handle more than one function on the hmesh, you create several structures of the type `F_HDATA2D` (`function_1`, `function_2`, ...) like above and add them to the hmesh by:

```
GRAPE (hmesh,"add-function")(&function_1);
GRAPE (hmesh,"add-function")(&function_2); ...
```

In our application we calculate the function values at the nodes and point based function error estimates in advance and store them in the `function_values` and `error_estimates` respectively.

### `data_access_function()`

The function specified by the pointer `f_hdata2d->f` is used to get the value(s) of a function at a given point. As can be seen above `f_hdata2d->f` points to the `data_access_function()`. The point for which the function value should be returned

is specified either by the element and local coordinates or by the element and a local vertex index. The resulting function value is stored in the argument array `result`.

---

```
static void data_access_function(HELEMENT2D* helement,
                                int           index,
                                double        coord[],
                                double        result[],
                                void*         function_data)
{
    if(coord){
        ...; /* calculate the function value at local coord
               coord[0,...,max]
               */
    }
    else{
        ...; /* calculate the function value at
               x = helement->vertex[i][0],
               y = helement->vertex[i][1],
               z = helement->vertex[i][2].
               */
    }
}

return;
}
```

---

As explained above we can access the nodal function values via the `function_data` argument. If `coord` is not `NULL` we apply a linear interpolation scheme.

```
static void data_access_function(HELEMENT2D* helement,
                                int           index,
                                double        coord[],
                                double        result[],
                                void*         function_data)
{
    double*       function_values;
    USER_FUNCTION_DATA* user_function_data;
    int i;

    user_function_data = (USER_FUNCTION_DATA*)function_data;
    function_values   = user_function_data->function_values;

    if(coord){
        *result = 0.0;
        for(i=0;i<3;i++)
            *result += coord[i] * function_values[helement->vindex[i]];
    }
    else
        *result = function_values[helement->vindex[index]];

    return;
}
```

### local\_information()

This routine provides element-dependent information about the function. Currently only the polynomial degree is supported.

---

```
static void local_information(HELEMENT2D* helement,
                             F_HEL_INFO2D* result,
                             void*         function_data)
{
    result->polynomial_degree = ...;

    return;
}
```

---

We choose to have only a piecewise linear function on our mesh so the polynomial degree is always 1.

```
static void local_information(HELEMENT2D* helement,
                             F_HEL_INFO2D* result,
                             void* function_data)
{
    result->polynomial_degree = 1;
    return;
}
```

### 13.9.1 get\_bounds()

This routine calculates the maximum and the minimum of the function on an element. In the mask we find:

---

```
static void get_bounds(HELEMENT2D* helement,
                      double* min,
                      double* max,
                      void* function_data)
{
    *min = ...;
    *max = ...;

    return;
}
```

---

And we adapt this to our piecewise linear function:

```
static void get_bounds(HELEMENT2D* helement,
                      double* min,
                      double* max,
                      void* function_data)
{
    USER_FUNCTION_DATA* user_function_data;
    double* function_values;

    int i;

    user_function_data = (USER_FUNCTION_DATA*)function_data;
    function_values = user_function_data->function_values;

    *min = function_values[helement->vindex[0]];
    *max = *min;

    for(i=0;i<2;i++){
        if(function_values[helement->vindex[i]] < *min)
            *min = function_values[helement->vindex[i]];
        if(function_values[helement->vindex[i]] > *max)
            *max = function_values[helement->vindex[i]];
    }

    return;
}
```

In the general case we refer to section 3 for the relation to error indicator values.

### get\_function\_vertex\_estimate()

The handling of function error estimators is analogous to the treatment of the geometry error estimators. We have stored the precalculated point estimators in the structure `f_hdata2d->function_data`. So this routine becomes:

```
static void get_function_vertex_estimate(HELEMENT2D* helement,
                                         double* results,
                                         void* function_data)
{
    USER_FUNCTION_DATA* user_function_data;
    double* error_estimates;

    int number_of_vertices = 3;
    int i;
```

```

user_function_data = (USER_FUNCTION_DATA*)function_data;
error_estimates    = user_function_data->error_estimates;

for(i=0;i<number_of_vertices;i++)
  results[i] = error_estimates[helement->vindex[i]];

return;
}

```

### get\_function\_element\_estimate()

Again we have basically the same code as in `get_geometry_element_estimate`.

```

static double get_function_element_estimate(HELEMENT2D* helement,
                                         void*           function_data)
{
  USER_FUNCTION_DATA* user_function_data;
  double*            error_estimates;

  user_function_data = (USER_FUNCTION_DATA*)function_data;
  error_estimates    = user_function_data->error_estimates;

  return error_estimates[((INTERFACE_ELEMENT*)helement)->belement->newnode];
}

```

## 14 Fundamental Methods on Hierarchical Meshes

In this chapter the methods on procedural hierarchical interfaces as they are implemented in the software environment **GRAPE** are discussed in detail. This documentation only represents the current state of development. Continuously new methods are included in the library.

### 14.1 GenMesh

#### 14.1.1 Memory Methods on GenMesh

**(GENMESH \*)GRAPE(GenMesh, "new-instance")(name)**

**CLASS \*GenMesh char \*name**

This method creates a new instance of class **GenMesh** with name **name** by allocating the necessary memory. The **current\_function**-pointer of the created **GenMesh** is initialized and the zero-function is set to be the actual function.

**(GENMESH \*)GRAPE(mesh, "free")()**

**GENMESH \*mesh**

This method frees the instance **mesh** by deleting the structures allocated by **new-instance**, that means the **GLIST \*current\_function**, the list of functions in **mesh→f\_data** and the instance itself.

**(GENMESH \*)GRAPE(mesh, "softcopy")(copy)**

**GENMESH \*mesh,\*copy**

Copies the instance **mesh** into **copy** by calling method on superclass and filling additional structure-pointers and variables, in particular the function-selector and the list of functions are copied. **copy** is returned.

**(GENMESH \*)GRAPE(mesh, "get-object")(obj,t)**

**GENMESH \*mesh,\*obj**  
**double t**

This method softcopies the instance **mesh** and its function-selector into the pointer **obj** and sets the time of **obj** to **t**. This **obj** is returned.

**(GENMESH \*)GRAPE(mesh, "copy-functions")(copy)**

**GENMESH \*mesh,\*copy**

This method hardcopies the **f\_data** list of the instance **mesh** to **copy**. If there are already some functions present in **copy**'s **f\_data**-list, these are removed. The instance **mesh** is returned completely unchanged.

**(GENMESH \*)GRAPE(mesh, "copy-function-selector")(copy)**

**GENMESH \*mesh,\*copy**

The function-selector of instance **copy** is deleted and the one of instance **mesh** is copied completely to **copy**. The instance **mesh** is returned unchanged.

**(GENMESH \*)GRAPE(mesh, "clear-function-selector")()**

**GENMESH \*mesh**

This method clears every slot in the list **current\_function** of instance **mesh**.

#### 14.1.2 Other Methods on GenMesh

**GRAPE(mesh, "xdr")(xdr)**

**XDR \*xdr**

This method implements XDR interface of **GRAPE** for the class **GenMesh**. To use the read/write facilities of **GRAPE**, one has to overload this method.

**(GLIST \*)GRAPE(mesh, "get-function-list")()**

**GENMESH \*mesh**

The list **f\_data** of instance **mesh** is copied into a **GLIST \*** and this is returned. The method recognizes eventual occurrence of circular lists and corrects it.

**(GENMESH\_FDATA \*)GRAPE(mesh, "get-function")(create, preferred\_slot, next\_slot, ..., least\_slot, NULL)**

**GENMESH \*mesh**  
**char \*create, \*preferred\_slot, \*next\_slot, \*least\_slot**

This method fulfils two tasks: First a new slot is created with name given by the string **create**, if this is not **NULL** and a slot with that name does not already exist. Second the slots of **mesh**'s function selector given by the following parameter-strings are inspected one by one. If a function has been selected for one of these slots, the data of this function is copied into the return parameter.

**(GENMESH \*)GRAPE(mesh, "select-function")(slot,function)**

```
GENMESH *mesh
char *slot,*function
```

This method selects **function** for the given **slot**. If the **slot** does not exist, it is created.

**(GENMESH \*)GRAPE(mesh, "get-info")(list)**

```
GENMESH *mesh
XLIST *list
```

Generates **list** with structure-dependent information of **mesh** like all GenMesh-entries and names of all functions in **mesh**'s **f\_data-list**.

**(GENMESH \*)GRAPE(mesh, "get-access-mode")(flags)**

```
GENMESH *mesh
MESH_ACCESS_FLAGS *flags
```

This method gets the actual **access\_mode** entry of instance **mesh**. Valid values are combinations of predefined flags **mafNone**, **mafSorted** and **mafBoundary**. The result is returned in **flags**.

**(GENMESH \*)GRAPE(mesh, "set-access-mode")(flags)**

```
GENMESH *mesh
MESH_ACCESS_FLAGS *flags
```

This method sets the parameter **flags** as actual **access\_mode** entry of instance **mesh**. Valid values are combinations of predefined flags **mafNone**, **mafSorted** and **mafBoundary**.

**(GENMESH \*)GRAPE(mesh, "get-picked-point")(element,coord,point,key)**

```
GENMESH *mesh
void **element
double *coord
VEC3 *point
int *key
```

This method allows picking a point either directly in the output-window by clicking left mouse-button or pressing 'p' while pointing on the output-window. In this case a layer for entering the point's coordinates pops up. This point is then searched in the given mesh by calling the appropriate search-method. All parameters are return-parameters: **point** contains the world-coordinates of the picked point, **key** contains 1001 if left mouse-button was pressed, 1002 or 1003 for other mouse-buttons and 'p' if this key was pressed. The element that contains the picked point and its local coordinates are returned in **element** and **coord**.

## 14.2 Mesh2d

### 14.2.1 Memory Methods on Mesh2d

**(MESH2D \*)GRAPE(Mesh2d, "new-instance")(name)**

```
CLASS *Mesh2d
char *name
```

This method creates a new instance of class `Mesh2d` with name `name` and allocates the necessary memory.

```
(MESH2D *)GRAPE(mesh, "free")()
    MESH2D *mesh
```

Just calls identical method on superclass `GenMesh`.

```
(MESH2D *)GRAPE(mesh, "softcopy")(copy)
    MESH2D *mesh,*copy
```

Copies the instance `mesh` into `copy` by calling the method on the superclass and additionally fills structure-pointers in `copy`, `copy` is returned.

```
(MESH2D *)GRAPE(mesh, "add-function")(f_data)
    MESH2D *mesh
    F_DATA2D *f_data
```

Adds the function given by `f_data` by copying and inserting it as first entry in the linked list given by `mesh→f_data`. Finally the function-selector is updated.

```
(MESH2D *)GRAPE(mesh, "remove-function")(name)
    MESH2D *mesh
    char *name
```

This method frees the first entry of the linked list `mesh→f_data` which corresponds to the given string `name`. The function-selector is updated afterwards.

#### 14.2.2 Display Methods on Mesh2d

```
(MESH2D *)GRAPE(mesh, "display")()
    MESH2D *mesh
```

The `Mesh2d` instance `mesh` is rendered on the standard device in a default manner. In patch and texture mode the elements are displayed as flat shaded patches, in grid mode the edges of the elements are drawn.

```
(MESH2D *)GRAPE(mesh, "isoline")(select, highscal, colorbar)
(MESH2D *)GRAPE(mesh, "isoline-disp")()
(MESH2D *)GRAPE(mesh, "isoline-select-disp")()

    MESH2D *mesh
    int select
    double highscal
    COLORBAR* colorbar
```

These two display-methods draw isolines and iso-patches on the `mesh`. For having nearly the same functionality the drawing is done in both cases by one method "isolines" just differing in the flag `select`: `select=0` means call from "isoline-disp", `select=1` indicates call from "isoline-select-disp". In grid-mode "isoline-disp" draws equidistant isolines with respect to the `colorbar`, that means line-distance, colors and min/max-values are taken from `colorbar`. "isoline-select" just draws selected isolines. In patch-mode iso-patches are drawn, the select-display version additionally draws the selected lines. Texture mode offers the same functionality as the `HMesh2d`-version, see 14.3.2 for details. All modes additionally allow to extrude the mesh in z-direction according to the function-values. By this a simultaneous function-graph/isoline-display can be obtained.

```
(MESH2D *)GRAPE(mesh, "vect")(double length, double width, double dist,
    double color_scal, double offset_scal, double min, double max)
(MESH2D *)GRAPE(mesh, "vect-disp")()
MESH2D *mesh
```

These methods offer the same functionality as the `Fe2d`-analogon.

```
(MESH2D *)GRAPE(mesh, "function-graph-disp")()
MESH2D *mesh
```

This method draws the instance `mesh` not as a plain mesh but the nodal z-coordinates are shifted by a multiple of the nodal function-values. The factor of this multiple can be changed in the options-layer for this display-method.

```
(MESH2D *)GRAPE(mesh, "geometry-graph-disp")()
MESH2D *mesh
```

Similar to the "isoline-disp" method, this method also draws isolines/iso-patches on the `mesh`. Instead of corresponding to the current function, these isolines belong to the z-coordinate-function of the mesh. Thus real geometry-isolines are drawn. This method is faster than using the function-selector and choosing the function z-coordinate, which gives the same result.

```
(MESH2D *)GRAPE(mesh, "shrink-disp")()
MESH2D *mesh
```

Similar to the standard display-method this one draws each element by a shrinked patch or shrinked polygon depending on the grid-patch mode. The shrink-factor can be changed by a corresponding ruler.

#### 14.2.3 Other Methods on Mesh2d

```
GRAPE(mesh, "xdr")(xdr)
XDR *xdr
```

This method implements XDR interface of **GRAPE** for the class **Mesh2d**. Of course neither function pointers nor the underlying user data can be handled, but all scalar values are converted, so it is an abstract method. To use the read/write facilities of **GRAPE**, one has to overload this method.

**(MESH2D \*)GRAPE(mesh, "get-info")(list)**

**MESH2D \*mesh**  
**XLIST \*list**

Fills **list** with structure-dependent information-strings. Actually it calls "**GenMesh-get-info**" and adds the information **mesh**→**dimension\_of\_world**.

**(GLIST \*)GRAPE(mesh, "get-function-list")()**

**MESH2D \*mesh**

Generates list of functions including default-functions.

**(MESH2D \*)GRAPE(mesh, "get-min-max")(pmin,pmax)**

**MESH2D \*mesh**  
**double \*pmin,pmax**

Evaluates the current data-function in all nodes of the **mesh** and determines the extremal values of these.

**(MESH2D \*)GRAPE(mesh, "value-min-max-send")()**

**MESH2D \*mesh**

This method calls "**get-min-max**" and prints the values to **stdout**.

**(MESH2D \*)GRAPE(mesh, "get-bnd-box")(xmin,xmax,ymin,ymax,zmin,zmax)**

**MESH2D \*mesh**  
**double \*xmin,\*xmax,\*ymin,\*ymax,\*zmin,\*zmax**

Determines the bounding-box of the instance **mesh** and returns it by its extremal coordinates.

**(MESH2D \*)GRAPE(mesh, "search")(v,c,el)**

**MESH2D \*mesh**  
**VEC3 v**  
**double \*c**  
**ELEMENT2D \*\*el**

This method searches within a **mesh** for a given point with world-coordinates **v**. The search applies neighbourhood-relationship if this is provided by an **element**→**descr**→**neighbour** function. Otherwise it uses straightforward mesh-traversal. If successful, the method returns in **el** the element which contains the given point and in **c** the local coordinates of this point. The pointer **c** has to be allocated with enough memory before calling the method. If the search failed, **el** will be **NULL**.

(MESH2D \*)GRAPE(mesh, "find-pick-intersec")(v,u,w,el,c)

```
MESH2D *mesh
VEC3 v
double u,w
ELEMENT2D **el
double *c
```

For a mesh with `dimension_of_world` equal 2 this method determines for given screen-coordinates `u,w` which point on the mesh is hit by these pick-coordinates. The world coordinates of this point are returned in `v`, the element which contains this point and its local-coordinates are returned in `el` and `c`.

### 14.3 HMesh2d

#### 14.3.1 Memory Methods on HMmesh2d

(HMESH2D \*)GRAPE(HMesh2d, "new-instance")(name)

```
CLASS *HMmesh2d char *name
```

This method creates a new instance of class HMmesh2d with name `name` and allocates the necessary memory. The function pointers `first_element` and `next_element` are set to default procedures.

(MESH2D \*)GRAPE(hmesh, "softcopy")(copy)

```
HMesh2D *hmesh,*copy
```

Copies the instance `hmesh` by calling the method on the superclass Mesh2d and fills additional structure-pointers in `copy`. `copy` is returned.

(HMESH2D \*)GRAPE(hmesh, "add-function")(f\_hdata)

```
HMesh2D *hmesh
F_HDATA2D *f_hdata
```

Adds the function given by `f_hdata` by copying and inserting it as first entry in the linked list given by `hmesh→f_data`. Finally the function-selector is updated.

(HMESH2D \*)GRAPE(hmesh, "remove-function")(name)

```
HMesh2D *hmesh
char *name
```

This method frees the first entry of the linked list `hmesh→f_data` which corresponds to the given string `name`. The function-selector is updated afterwards.

(HMESH2D \*)GRAPE(hmesh, "get-object")(obj,t)

```
HMesh2D *hmesh,*obj
double t
```

This method returns the instance `hmesh` at time `t`. This happens by softcopying `hmesh` and its function-selector to the pointer `obj`, setting `obj`'s time to `t` and returning `obj`. If `obj` already is filled, its `level_of_interest` and `threshold` are kept. (To be precise: `hmesh`'s parameters are set to these values.)

**(HMESH2D \*)GRAPE(hmesh, "hmesh-interactive-send")()**

**HMesh2d \*hmesh**

This method sets up the project *uif-hm2*, see 14.4.

**(HMESH2D \*)GRAPE(hmesh, "calc-lenestimators")(lens)**

**HMesh2d \*hmesh**

**HMesh2d\_Lens \*lens**

This method calculates lens-estimators for all nodes of the **hmesh** up to the leaf-level and returning them in **lens→lens\_data**. The calculation of the estimators is based on the distance of parent-nodes to child-nodes. These estimators are saturated by a blow-up procedure assuring that estimators of nodes of an element are bigger than all nodes of the elements's children. It is necessary that elements have access to their global vertex-indices in **vindex**. If this is not the case all estimators are set to zero.

#### 14.3.2 Display Methods on HMesh2d

All display-methods on HMeshes offer the possibility of adaptive visualization with regard to various thresholds and error estimates which have to be provided by the user in a prescribed way or can also for example in case of lens-estimators be calculated by default-routines. See the example-HMesh-interfaces for further details. Without any error-estimates the HMesh is always drawn up to the level given by its **level\_of\_interest** entry, whereas with error-estimates some elements may be extracted on a higher hierarchy-level without losing essential detail-information but gaining a speedup because of this partial inspection of the mesh.

**(HMESH2D \*)GRAPE(hmesh, "display")()**

**HMesh2d \*hmesh**

The HMesh2d instance **hmesh** is rendered on the standard device in a default manner. In patch and texture mode the elements are displayed as flat shaded patches, in grid mode the edges of the elements are drawn. If geometry-estimators or lens-estimators are provided, these are used for adaptive visualization. An options-layer allows to activate/deactivate the lens-adaptivity.

**(HMESH2D \*)GRAPE(hmesh, "vect-disp")()**

**HMesh2d \*hmesh**

This method draws a 2d or 3d vector-valued function as arrow-field on the mesh. The mesh is drawn in a default manner, the arrows as flat arrows in 3d-space starting from points of a virtual equidistant grid put on the instance **hmesh**.

The options-layer shown in Figure 24 allows changing diplay-parameters.

The ruler **distance** determines the z-shift of all arrows, the **mesh-width** is the distance between the arrows. **Length** gives a scale-factor which is applied to each function-value-vector. If the resulting length is bigger than the value **max\_length**, the vector is shortened to this value. If lens-estimators are provided, a lens can be activated/deactivated and a final ruler called **lens\_factor** allows to enlargen the mesh-width automatically in case of activated lens. An example is shown in figure 25.

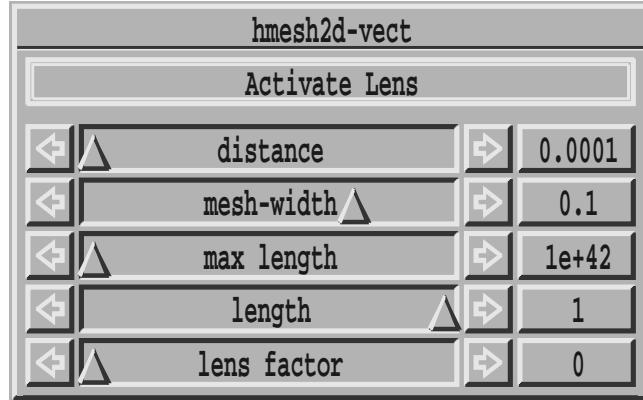


Figure 24: Options Layer for HMmesh2d-vec-disp

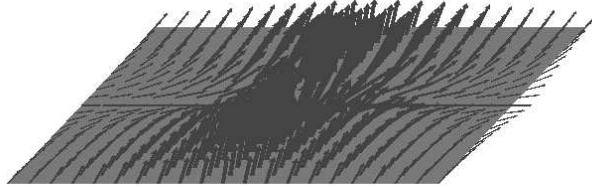


Figure 25: example of "vect-disp"

```
(HMESH2D *)GRAPE(hmesh, "rgb-disp")()
HMESH2D *hmesh
```

This method draws vector-valued functions encoding the data in rgb-space. This is done by cutting the function-vectors componentwise to the unit-interval and taking these values as rgb-vector. Every element is patched with these colors in its vertices. A lens is automatically initialized based on the estimates generated by the method "calc-lenstimators". Grid-mode is not supported. An example is shown in Figure 26.

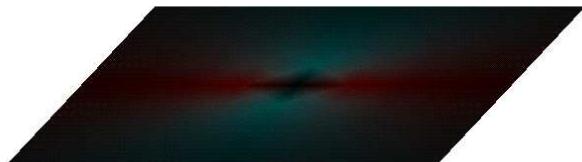


Figure 26: Example of "rgb-disp"

**(HMESH2D \*)GRAPE(hmesh, "shrink-disp")()**

**HMESH2D \*hmesh**

Similar to the standard display-method this one draws each element by a shrinked patch or shrinked polygon depending on grid-patch mode. The shrink-factor can be changed by a corresponding ruler. Additional a lens can be activated/deactivated. An example is shown in Figure 27.

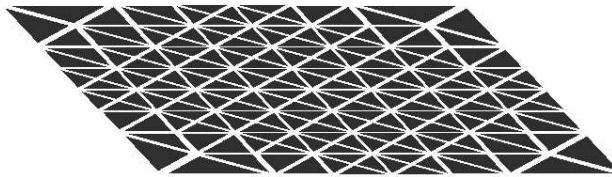


Figure 27: example of "shrink-disp"

**(HMESH2D \*)GRAPE(hmesh, "geometry-graph-disp")()**

**HMESH2D \*hmesh**

Similar to the "isoline-disp" method, this method also draws isolines/isopatches on the hmesh. Instead of corresponding to the current function, these isolines belong to the z-coordinate-function of the hmesh. Thus real geometry-isolines are drawn. This method is faster than using the function-selector and choosing the function z-coordinate, which gives the same result. With the options-layer shown in Figure 28 a lens can be activated/deactivated, the display can be switched to a projection on the x/y-plane and the colorbar can be accessed. An example is shown in Figure 29.

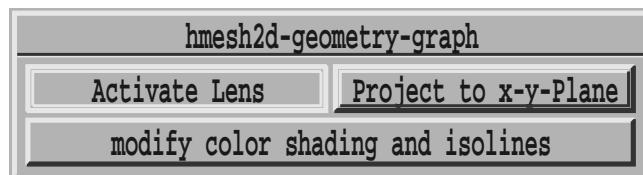


Figure 28: Options Layer for HMmesh2d-geometry-graph-disp

**(HMESH2D \*)GRAPE(hmesh, "function-graph-disp")()**

**HMESH2D \*hmesh**

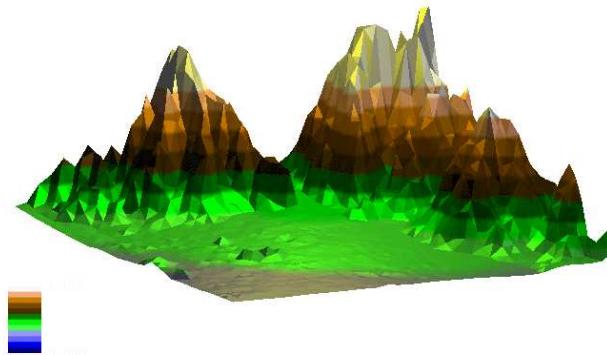


Figure 29: example of "geometry-graph-disp"

This method draws the instance `hmesh` not as a plain mesh but the nodal z-coordinates are shifted by a multiple of the nodal function-values. The factor of this multiple can be changed as `graph-height` in the options-layer for this display-method shown in figure 30. A lens can be activated/deactivated if lens-estimators are supported in the instance `hmesh`. An example is shown in figure 31.

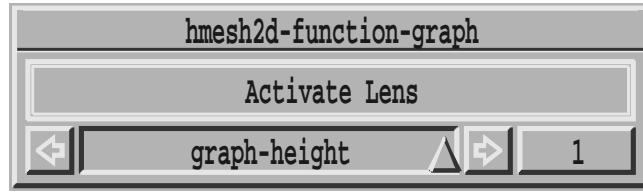


Figure 30: Options Layer for HMESH2d-geometry-graph-disp

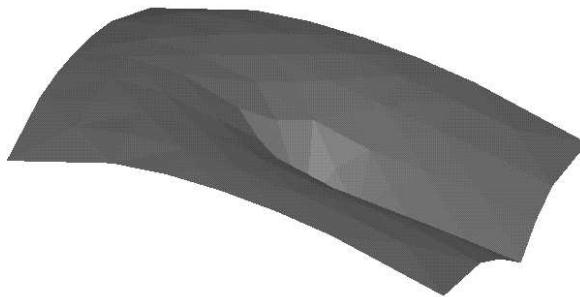


Figure 31: example of "function-graph-disp"

```
(HMESH2D *)GRAPE(hmesh, "isoline-disp")()
(HMESH2D *)GRAPE(hmesh, "isoline-select-disp")()
HMESH2D *hmesh
```

These two display-methods draw isolines and iso-patches on the mesh. In grid-mode "isoline-disp" draws equidistant isolines with respect to the actual colorbar, that means line-distance, colors and min/max-values are taken from the colorbar. "isoline-select-disp" just draws selected isolines. In patch-mode iso-patches are drawn, the select-display version additionally draws the selected lines. The following figure shows the options-layer for this method in texture mode. In other modes some buttons/rulers are hidden, if they are not used. In

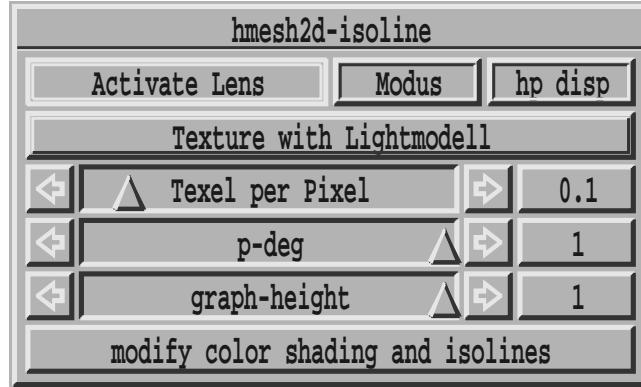


Figure 32: Options Layer for HMmesh2d-isoline-disp

texture-mode the surface of the mesh is patched with textures which are able to react on light-sources. This ability can be activated/deactivated by the button `Texture with lightmodel`. The button `modus` activates/deactivates the ruler `graph-height` and an additional extruding of the mesh in z-direction according to a multiple of the nodal function-values, the factor given by `graph-height`. The button `hp disp` activates the remaining two rulers `Texel per Pixel` and `p-deg`. This mode draws higher polynomial textures on each element. The polynomial degree and resolution of the textures are determined by these rulers. An example of this display-method is shown in Figure 33.

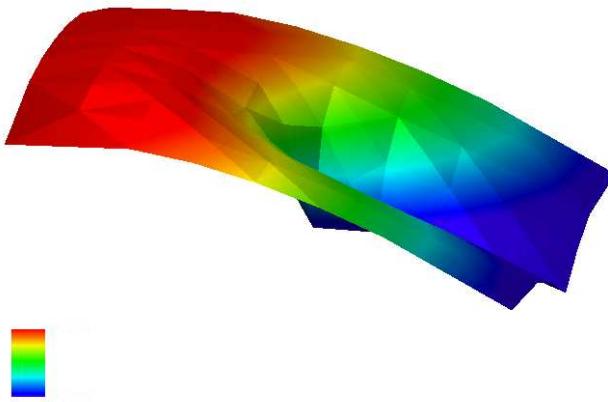


Figure 33: example of "isoline-disp"

```
(HMESH2D *)GRAPE(hmesh, "debug-disp")()
HMESH2D *hmesh
```

This method is a debug-tool for hmesh-interfaces. It displays the hmesh like the shrink-method and the element's boundaries by blue lines. Therefore the shrink-options-layer is shown. The traversal-functions of the hmesh-interface can now be called by pressing the corresponding buttons of the options-layer for this method shown in Figure 34.

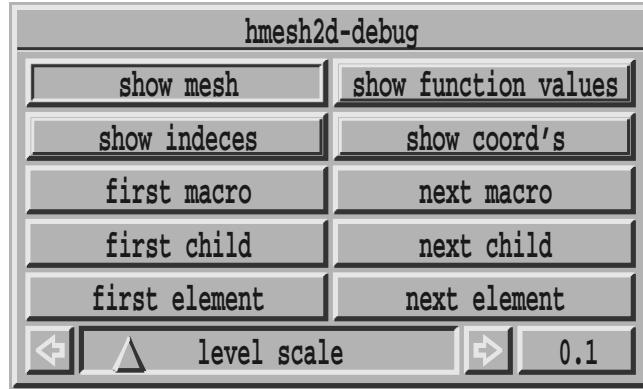


Figure 34: Options Layer for HMmesh2d-debug-disp

The resulting element and its parents-path to a macro-element is highlighted by white boundaries. Child-elements are shifted from their parents in z-direction according to the value of the ruler **level-scale**. Additional information is shown for the actual element by activating one of the buttons **show indices**, **show coord's** or **show function-values**. The drawing of the mesh can be enabled/disabled by the button **show mesh**. An example is shown in Figure 35.

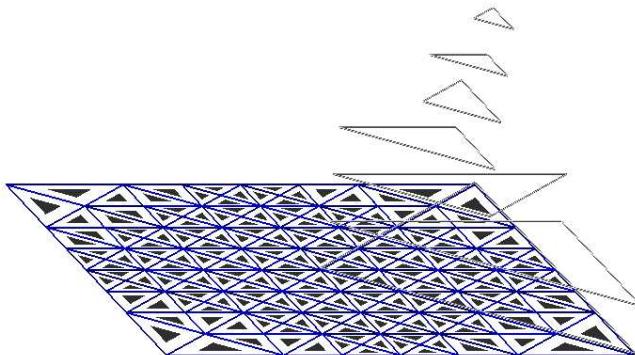


Figure 35: example of "debug-disp"

#### 14.3.3 Other Methods on HMmesh2d

**GRAPE(hmesh, "xdr")**(**xdr**)

**XDR \*xdr**

This method implements XDR interface of **GRAPE** for the class **HMmesh2d**. Of course neither function pointers nor the underlying user data can be handled, but all scalar values are converted, so it is an abstract method. To use the read/write facilities of **GRAPE**, one has to

overload this method.

**(HMESH2D \*)GRAPE(hmesh, "get-bnd-box")(*xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*)**

```
HMESH2D *hmesh
double *xmin,*xmax,*ymin,*ymax,*zmin,*zmax
```

Determines the bounding-box of the instance *mesh* and returns it by its extremal coordinates. This method is faster than the Mesh2d-version because only macro-elements have to be inspected.

**(HMESH2D \*)GRAPE(hmesh, "search")(*v*, *c*, *el*)**

```
HMESH2D *hmesh
VEC3 v
double *c
HELEMENT2D **el
```

This method searches within the *hmesh* for a given point with world-coordinates *v*. If *el* is not NULL local search is performed starting at the element *el*. This means walking up in the hierarchy until an element is found, which contains *v*, and then walking down to the wanted element. If *el* is NULL at the beginning, hierarchical search is applied by walking over the macro-elements and then walking down the hierarchy.

If successful, the method returns in *el* the element on level *level\_of\_interest* which contains the given point *v*, and in *c* the local coordinates of this point. The pointer *varc* has to be allocated with enough memory before calling the method. If the search failed, *el* will be NULL. The search relies on the HMESH2D-method "select\_child". On one hand this means that this pointer has to be filled in the interface at least with the default-method "g\_hmesh2d\_def\_select\_child". On the other hand it implies that implementing this method efficiently is a possibility of speeding up the search.

**(HMESH2D \*)GRAPE(hmesh, "find-pick-intersec")(*v,u,w,el,c*)**

```
HMESH2D *hmesh
VEC3 v
double u,w
HELEMENT2D **el
double *c
```

For *hmesh* with *dimension\_of\_world* equal 2 this method determines for given screen-coordinates *u,w* which point on the mesh is hit by these pick-coordinates. The world coordinates of this point are returned in *v*, the element on level *mesh*→*level\_of\_interest* which contains this point and its local-coordinates are returned in *el* and *c*. The only difference to the Mesh2d-version is the use of improved hierarchical searching.

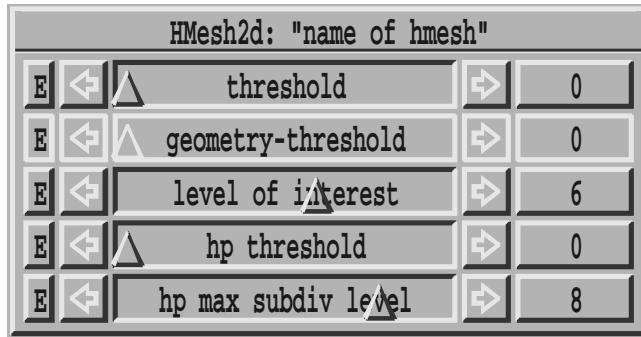
#### 14.4 Projects for Meshes

All meshes can be accessed interactively by the projects *uif-gm*, *uif-hm2* and *uif-hm3*. The first of these projects generates an options layer as in Figure 36. It is an interface for the function-

Figure 36: Layer for Project *uif-gm*

selector of the mesh. The leftmost button pops up a menu to select the slot, according to this the selected function automatically is displayed on the right button. This button also allows to change the selected function for the given slot by pressing it. The list of possible choices popping up after pressing contains the user-defined ones and additional some default-functions like coordinate-functions.

The project *uif-hm2* generates the same layer, but additional a second one to change the entries of a HMesh2d instance like **level-of-interest** and various threshold-values like **geometry-threshold**, **threshold** and **hp-threshold** for adaptive visualization with regard to the geometry, the function-values and polynomial-degree. This layer is shown in Figure 37.

Figure 37: Additional Layer for Project *uif-hm2*

#### 14.5 Methods on other classes related to meshes

**(HMESH2D\_LENS \*)GRAPE(lens, "get-lens")(name)**

```
HMESH2D_LENS *lens
char *name
```

This method creates a new instance of Class `Hmesh2d_Lens` with name given as parameter `name` and initializes it with default values. In particular the estimators and thresholds are set to default-functions. The lens is assigned to an own interactive layer which allows to manipulate the lens-parameters.

**(HMESH2D\_LENS \*)GRAPE(lens, "display")()**

```
HMESH2D_LENS *lens
```

This method draws an icosaeder related to the parameters of the `lens` as a wire-frame-model. The center is the entry `coord` of the instance `lens`. The radius is a convex-combination depending on `show_threshold` between the inner and outer region of the mesh.

```
(MESH2D *)GRAPE(sc, "convert-to-mesh-send")()
SCENE *sc
```

This method tries to convert the instance `sc->object` to one of class `Mesh2d`. If successful, this new instance is taken as `sc`'s object. The old structure is still accessible via the created mesh's `user_data` pointer.

```
(MESH2D *)GRAPE(tr, "convert-to-mesh2d")()
TRIANG2D *tr
```

This method generates a `Mesh2d` instance based on the data of `tr`. Especially the original `Triang2d` structure remains unchanged and is accessible from the returned mesh via its `user_data` pointer. The resulting mesh carries no functions.

```
(MESH2D *)GRAPE(fe, "convert-to-mesh2d")()
FE2D *fe
```

This method generates a `Mesh2d` instance based on the data of `fe`. Especially the original `Fe2d` structure remains unchanged and is accessible from the returned mesh via its `user_data` pointer.

## 14.6 HMesh3d

### 14.6.1 Memory Methods on HMesh3d

```
(HMESH3D *)GRAPE(HMesh3d, "new-instance")(name)
CLASS *HMesh3d char *name
```

This method creates a new instance of class `HMesh3d` with name "name" and allocates the necessary memory. The function pointers `first_element` and `next_element` are set to default procedures.

```
GRAPE(hmesh, "free")()
HMESH3D *hmesh;
```

The associated memory of the instance variables is freed and the instance is deleted.

```
GRAPE(hmesh, "softcopy")(copy)
HMESH3D *hmesh;
HMESH3D *copy;
```

Make a softcopy of a `HMesh3d` instance. If `copy` is not NULL then this instance is used to store the copy otherwise a new `HMesh3d` instance is created.

```
(HMESH3D *)GRAPE(hmesh, "add-function")(f_hdata)
HMESH3D *hmesh
F_HDATA3D *f_hdata
```

Adds the function given by `f_hdata` by copying and inserting it as first entry in the linked list given by `hmesh→f_data`. Finally the function-selector is updated.

**(HMESH3D \*)GRAPE(hmesh, "remove-function")(name)**

```
HMESH3D *hmesh
char *name
```

This method frees the first entry of the linked list `hmesh→f_data` which corresponds to the given string `name`. The function-selector is updated afterwards.

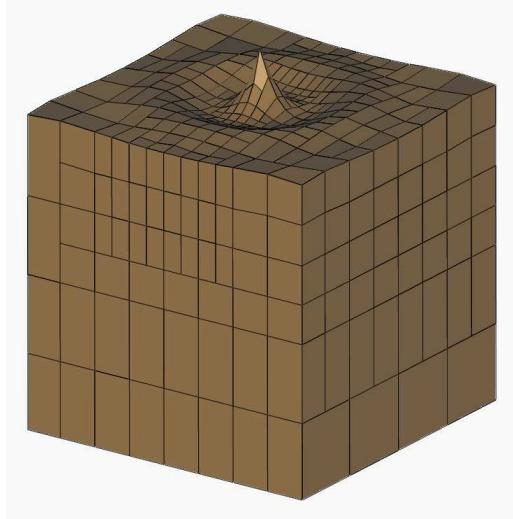


Figure 38: Adaptive visualization of the boundary of a domain.

**(HMESH3D \*)GRAPE(hmesh, "get-object")(obj,t)**

```
HMESH3D *hmesh,*obj
double t
```

This method returns the instance `hmesh` at time `t`. This happens by softcopying `hmesh` and its function-selector to the pointer `obj`, setting `obj`'s time to `t` and returning `obj`. If `obj` already is filled, its `level_of_interest` and `threshold` are kept. (To be precise: `hmesh`'s parameters are set to these values.)

#### 14.6.2 Display Methods on HMes3d

All display-methods on HMes3d offer the possibility of adaptive visualization with regard to various thresholds and error estimates which have to be provided by the user in a prescribed way or can be calculated by default-routines. See the example-HMesh-interfaces for further details. Without any error-estimates the HMesh is always drawn up to the level given by its `level_of_interest` entry, whereas with error-estimates some elements may be extracted on a higher hierarchy-level without losing essential detail-information but gaining a speedup because of this partial inspection of the mesh.

**(HMESH3D \*)GRAPE(hmesh, "display")()**

```
HMESH3D *hmesh
```

The HMesh3d instance hmesh is rendered on the standard device in a default manner. All the boundary faces of the elements are displayed as flat shaded patches in patch mode and in grid mode the edges of these faces are drawn. If boundary-estimators are provided, these are used for adaptive visualization.

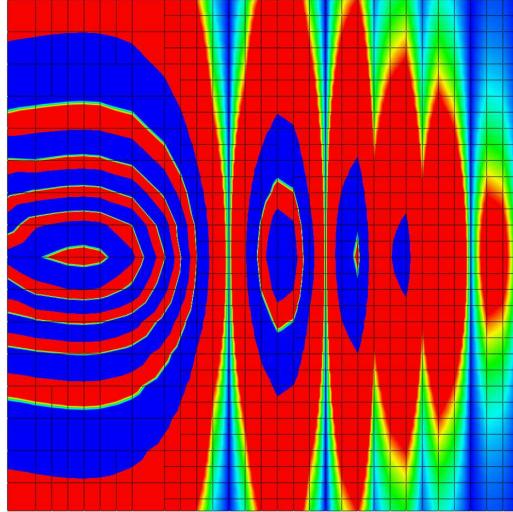


Figure 39: Colorshading on a slice.

```
(HMESH3D *)GRAPE(hmesh, "clip")(f, var, draw_clip, scal)
```

```
GRAPE(mesh, "clip-ball-disp")()
```

```
GRAPE(mesh, "clip-plane-disp")()
```

```
GRAPE(mesh, "clip-deg-fine-disp")()
```

```
GRAPE(mesh, "clip-isoline-disp")()
```

```
GRAPE(mesh, "clip-vect-disp")()
```

```
HMESH3D *hmesh
MESH3D *mesh
double (*f)(VEC3 x,void *var)
void *var
int (*draw_clip)(CLIPM3D_PAR)
void *scal
```

These methods are designed to support general clipping facilities with varying hyper surface and local rendering function. The most basic method with the greatest flexibility is the method "clip". The others are special predefined interactive methods derived from the latter one. The domain of the mesh is split by the implicitly given clipping function  $f$  into a visible ( $f > 0$ ) and an invisible ( $f < 0$ ) part. The zero level of this clipping function defines the intersection surface. The method "clip" on HMesh3d supports hierarchical search algorithms for extracting the intersection surface and uses adaptive resolution strategies, if the error estimator of the underlying function data is supplied.

The variable *var* in the parameter list of the method call and of this function *f* must be of the same type. The parameter *\*var* allows an interactive varying of the clipping function. The implicit functions describing a plane or a ball are predefined:

```
typedef struct MPLANE_PARM = { double n_x, n_y, n_z, distance;};

double mplane(VEC3 x, MPLANE_PARM *var)

typedef struct MBALL_PARM = { double cx,cy,cz,radius;};

double mball(VEC3 x, MBALL_PARM *var)
```

The pointer *draw\_clip* points to a function which renders the surface of intersection on a single element. This function works on the data corresponding to one single clipped element. A NIL pointer causes the standard visualization of the surfaces including normal shading.

There are some predefined local rendering functions:

- **draw\_clipm\_isoline**: generates an isoline image of a one dimensional FEM function or of the absolute value of a vector-valued function.
- **draw\_clipm\_deg\_fine**: shows the size of each clipped element in colour.
- **draw\_clipm\_vect**: draws a mesh of vector valued function values projected onto the tangent plane of the intersection surface, where the colour represents the component normal the tangent plane (only applicable if the function is three dimensional).

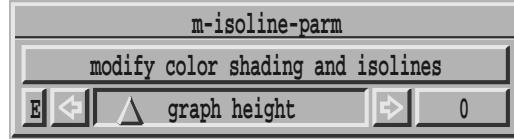


Figure 40: Interactive layer of the "clip-isoline-disp" method.

The interactive methods "clip-isoline-disp", "clip-deg-fine-disp", "clip-vect-disp" use these routines while calling the standard clipping method "clip". Figure 40 shows the interactive layer of the "clip-isoline-disp" method. With the rulers "normal x/y/z" the normal of a clipping plane is chosen, whereas the "distance" ruler adjusts the plane to the desired position. The ruler "graph height" gives a scaled height to the graph of the displayed isolines. Finally the "colorbar" interface can be used to adjust the colours of the isolines to the given function values.

To construct a new private rendering method working on the surface of intersection, the user can add new clipping methods by using other and different surfaces or by adding new local rendering functions. Therefore we now explain the parameter structure for the local rendering function **draw\_clip**. The general 'clip' method calls this function in the following way:

```
draw_clip((CLIPM3D_PAR *clip3d_par)
```

The structure *clip3d\_par* with the following *typedef* is filled by the general "clip" method.

```
typedef struct clipm3d_par
{ ELEMENT3D *e;
  int n;
  VEC3 *v;
```

```

double *coord;
double (*f)();
void *var;
void *scal;
int flag ;
int dimension_of_value;
} CLIPM3D_PAR;

```

This structure describes the intersection of the hyper surface with a single element. The filled structure is delivered by the actual method "clip". The variable *e* points to the intersected element and *n* gives the number of intersection points of one connected intersection patch. The array  $v[0 \dots n - 1][0 \dots 2]$  contains the global Euclidean coordinates of these points. They are located on edges of the element's boundary at a linearly interpolated intersection position. In all the listed interactive methods a plane is used as the clipping function. *coord* is a pointer to a list of the local coordinates of the intersection points , which are stored in global coordinates in *v*. In this 1D array the *i*'th coordinate vector starts at position (*coord* + *i* \* *d*) where *d* is the dimension of coordinate space on the current element. The pointer *var* points to the parameters of the function describing the intersecting hyper surface. At the memory address *scal* the function *draw\_clip* finds a user defined structure filled with parameters for the specific rendering style. These parameters can be influenced interactively, if the user supports this in an appropriate interactive method (see also the GRAPE reference manual).

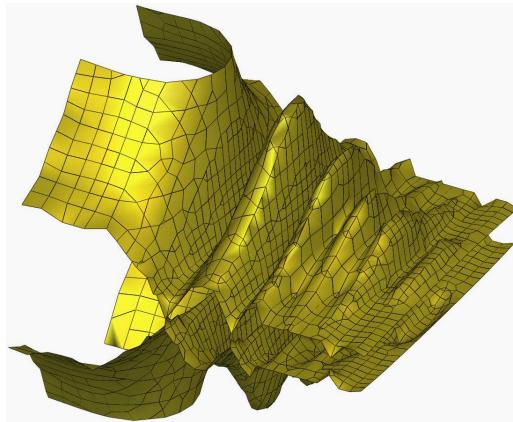


Figure 41: Adaptive visualization of an iso-surface.

(HMESH3D \*)GRAPE(hmesh, "level")(func, lev, mode)

GRAPE(mesh, "level-disp")()

```

HMESH3D *hmesh
MESH3D *mesh
void (*func)(ELEMENT3D *el, int, double *coord, double *val, void *function_data)
double lev
int mode

```

The slices of those elements where the one-dimensional finite element function *func* has the level *lev* are calculated and displayed. The parameters of *func* are equal to those used for a

one dimensional FEM function on a *HMesh3d* instance: *f(HELEMENT3D \*el, double \*coord, double \*val, void \*function\_data)*. If *func* is NULL and *hmesh* includes a function then this finite element function is taken into account. If its dimension of value is greater than 1 the absolute value of the value vector defines the function under consideration. The resulting 2D level surface consists of a polygonal approximation. If "mode = 1" or "mode = -1" the super or sub volumes of the function on the volume geometry are shown.

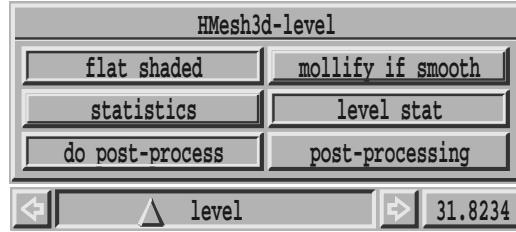


Figure 42: Interactive layer and ruler of the "level-disp" method.

The method "level-disp" calls the method "level" with the standard finite element function and inserts a ruler (cf. Figure 42) into the options menu to adjust the value "lev" interactively. The method "level" on *HMesh3d* supports hierarchical and adaptive search algorithms, if the error estimators of the underlying function data is supplied. Furthermore an interactive layer with the following buttons is added:

- **flat shaded** Turns on/off the flat shading mode.
- **mollify if smooth** If the flat shading mode is turned off the extracted grid of the isosurface is smoothed by an mollify algorithm.
- **statistics** Prints the threshold value and the number of actually rendered elements into the graphic window. Three rulers to adjust the position of the information in the graphic window appear.
- **level stat** Writes some information about the hierarchical and adaptive extraction of the isosurface onto the standard output.
- **do post-process** Calls the routine "g\_PostProcessCurrSurfMesh" if "flat shaded" is turned off. If this routine is selected the extracted surface mesh is smoothed and small cracks in the surface are closed. Such cracks may occure in the case of hexahedrons, where data is interpolated trilinear.
- **post-processing** Calls the method "post-processing-send", which is described below.

**(HMESH3D \*)GRAPE(hmesh, "post-processing-send")()**

**HMESH3D \*hmesh**

The method "post-processing-send" inserts a post processing layer into the options menu, which enables the user to activate one or more cutting planes, which cut off certain areas of the isosurface (cf. Figure 43). In detail there is the following functionality:

- **edit** Inserts the interfaces of the actual cutting planes.
- **add plane** Adds a cutting plane. An interface to adjust the position of the plane appears.

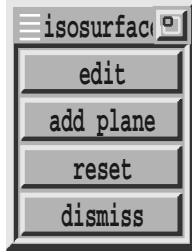


Figure 43: The interface inserted by the method "post-processing-send".

- **reset** Resets to a default status.
- **dismiss** Closes the interactive interface.

**GRAPE(hmesh,"shrink-disp")()**

**HMESH3D\* hmesh**

This method draws all elements of the mesh shrinked by a factor which can be changed by the corresponding ruler. The elements are drawn as a grid or as patches depending on the `grid-patch` mode. The method uses the `threshold` and `get_element_estimate` of the meshes `f_data` to determine whether it displays elements on a coarser level.

**(HMESH3D\*) GRAPE(hmesh,"inspect-disp")()**

**HMESH3D\* hmesh**

The "inspect-disp" method works similar to its twodimensional pendant. It displays the complete mesh with shrinked elements (as grid or as patches) and a grid of the boundary-faces of the mesh using blue lines. It is possible to traverse the mesh hierarchically via the buttons of the group `select element to inspect`.

Inspect features two different traversal modes, which can be selected via the `use function threshold` button: When the checkbox is turned off, the user can traverse the whole hierarchical mesh and the grid is displayed on the level of the current element. When turned on, the `threshold` of `f_data` and the current `level_of_interest` select the subtree which can be traversed and is displayed. The boundary error estimator is used to draw the blue boundary grid.

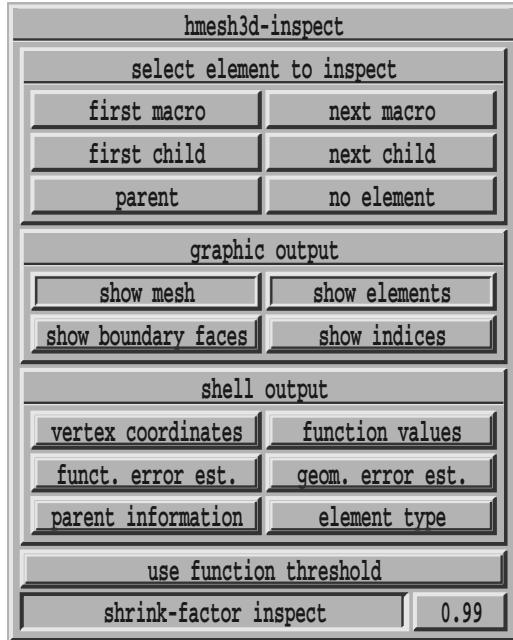
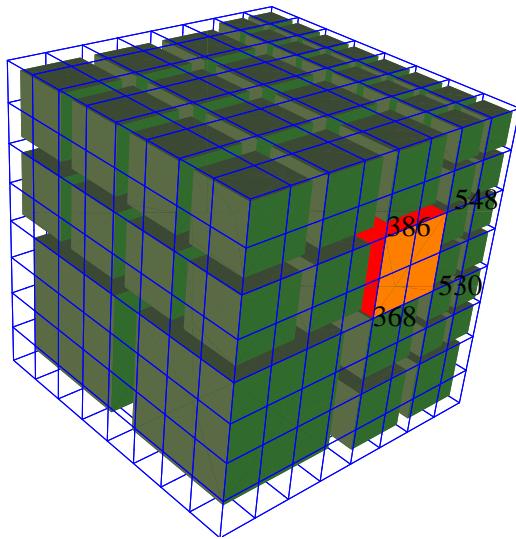
The current element is always highlighted in red. Additional debugging-information can be shown in the graphic window or in the shell: Boundary faces can be marked with orange color and global vertex indices can be shown in the graphics window. It is also possible to print some information about the current element and the function on it to `stderr`.

#### 14.6.3 Boundary Display Methods on HMESH3D

**GRAPE(hmesh, "bnd-isoline-disp")()**

**GRAPE(mesh, "bnd-isoline-select-disp")()**

**HMESH3D \*hmesh**

Figure 44: The options layer for `inspect-disp`.Figure 45: An example of `inspect-disp`.

These methods produce an isoline image of a one dimensional FEM function (or of the absolute value of a vector valued function) at the boundary of the *HMesh3d*-Instance. The method *bnd-isoline-select-disp* displays only those isolines whose values are stored in the array *colorbar->values*. This method also supports the texture mode of GRAPE making it possible to use a light model together with the use of colors.

The method works especially efficient if the used mesh supports an explicit boundary traversal. To this end there are flags in the *Mesh3d* struct (the super-class of *HMesh3d*) to indicate the currently set traversal mode (*access\_mode*) and the supported traversal modes (*access\_capability*).

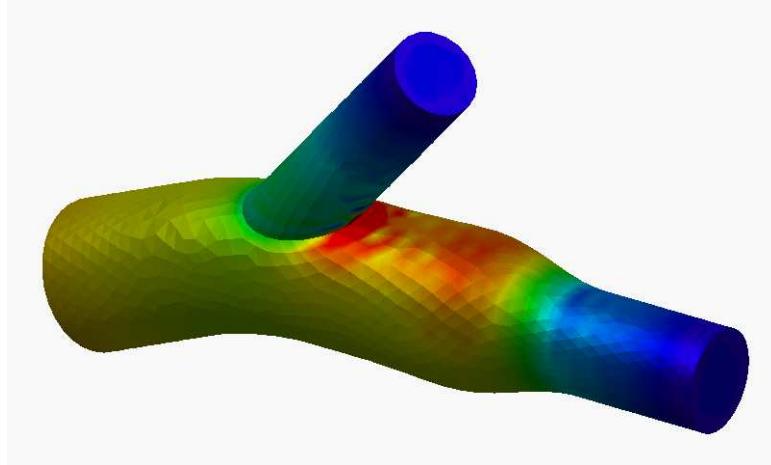


Figure 46: Colorshading of the pressure at the surface of an artery.

```
typedef struct mesh3d{
    ...
    MESH_ACCESS_FLAGS access_mode;
    MESH_ACCESS_FLAGS access_capability;
} MESH3D;
```

It is the user's responsibility to set the flag *access\_capability* and to program the mesh traversal routines (*first\_element*, *next\_element*, etc.) so that the set access capabilities are indeed supported. At the moment there are only two possible access flags for different types of mesh traversal:

```
typedef enum {
    mafNone          = 0,
    mafSorted        = 1<<0,
    mafBoundary      = 1<<1
} MESH_ACCESS_FLAGS;
```

If the method *bnd-isoline-disp* can set the flag *access\_mode* to *mafBoundary* then it will not be checked again if the element is really part of the boundary. Otherwise the method checks each element during the mesh traversal.

Setting or testing the access flags should be done with the methods

```
GRAPE(hmesh, "get-access-mode")(flags)
GENMESH *hmesh
MESH_ACCESS_FLAGS *flags
```

```
GRAPE(hmesh, "set-access-mode")(flags)
GENMESH *hmesh
MESH_ACCESS_FLAGS flags
```

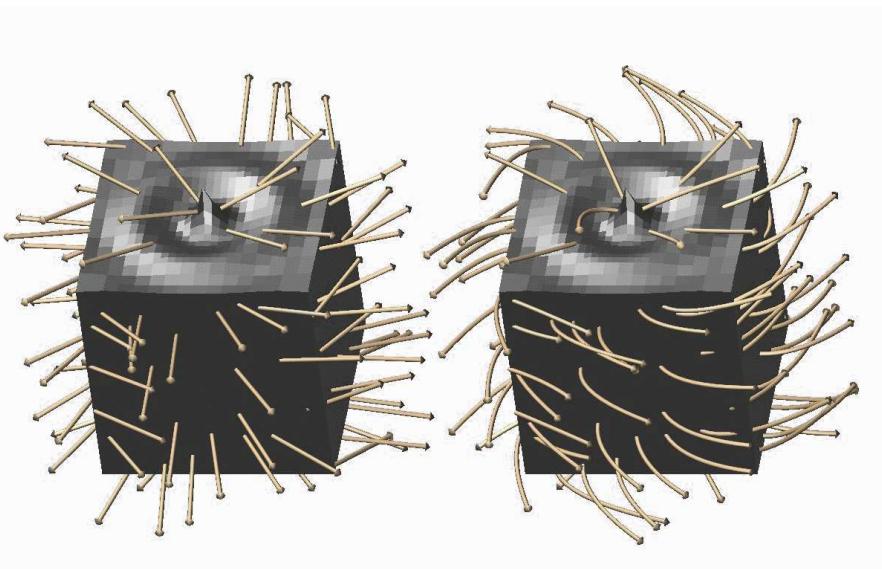


Figure 47: First and second order visualization of a vector field at the boundary of an object.

**GRAPE(hmesh, "bnd-vect-disp")()**

**HMESH3D \*hmesh**

This method visualizes the data on the boundary of an *HMesh3d* object by drawing random-distributed arrows. If the dimension of the data is less than 3 then the method draws vectors which are orthogonal on the boundary and whose length indicates the absolute value of the data. If the dimension of the data is greater than 3 a warning is printed and the first 3 coordinates of the data are used for the display; it is always possible to force the use of vectors orthogonal to the boundary by pressing the button "Use Normal Vectors". By pressing the button "Use Second Order" curved arrows are drawn giving a second order approximation for the data.

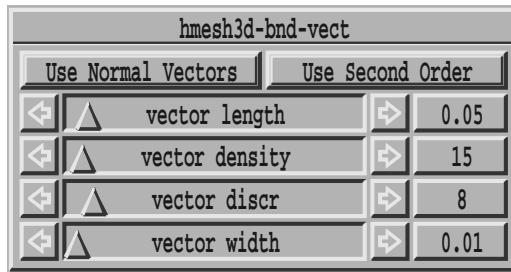


Figure 48: Interactive layer of the "bnd-vect-disp" method when the second order mode is not used.

Description of the rulers:

- "vector length" chooses the length of the drawn vectors.
- "vector density" affects the number of arrows drawn.
- "vector discr" determines the discretization level of the drawn arrows (e.g. 1 = line, 2 = flat arrows etc.)
- "vector width" chooses the thickness of the drawn arrows.

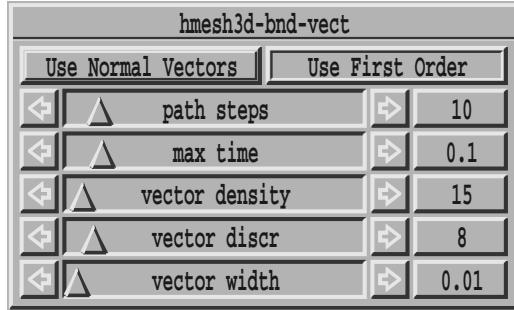


Figure 49: Interactive layer of the "bnd-vec-disp" method when the second order mode is used.

New rulers:

- "path steps" chooses the number of straight segments with which the second order approximation is drawn.
- "max time" determines the end time of the path of a particle moving along the vector field; since the drawn arrow represents a second order approximation to that path, this ruler controls the length of the arrow.

#### 14.6.4 I/O Methods on HMesh3d

**GRAPE(hmesh, "xdr")**(xdr)

XDR \*xdrp

This method implements XDR interface of **GRAPE** for the class **HMesh3d**. Of course neither function pointers nor the underlying user data can be handled, but all scalar values are converted, so it is an abstract method. To use the read/write facilities of **GRAPE**, one has to overload this method.

#### 14.6.5 Other Methods on HM3d

(HMESH3D \*)**GRAPE(hmesh, "get-bnd-box")**(xmin,xmax,ymin,ymax,zmin,zmax)

**HMESH3D \*hmesh**

**double \*xmin,\*xmax,\*ymin,\*ymax,\*zmin,\*zmax**

Determines the bounding-box of the instance **mesh** and returns it by its extremal coordinates. This method is faster than the **Mesh3d**-version because only macro-elements have to be inspected.

(HMESH3D \*)**GRAPE(hmesh, "search")**(v, c, el)

**HMESH3D \*hmesh**

**VEC3 v**

**double \*c**

**HELEMENT2D \*\*el**

This method searches within the `hmesh` for a given point with world-coordinates `v`. If `e1` is not NULL local search is performed starting at the element `e1`. This means walking up in the hierarchy until an element is found, which contains `v`, and then walking down to the wanted element. If `e1` is NULL at the beginning, hierarchical search is applied by walking over the macro-elements and then walking down the hierarchy.

If successful, the method returns in `e1` the element on level `level_of_interest` which contains the given point `v`, and in `c` the local coordinates of this point. The pointer `varc` has to be allocated with enough memory before calling the method. If the search failed, `e1` will be NULL. The search relies on the `HMesh3d`-method "select\_child". On one hand this means that this pointer has to be filled in the interface at least with the default-method "`g_hmesh3d_def_select_child`". On the other hand it implies that implementing this method efficiently is a possibility of speeding up the search.

`(HMESH3D *)GRAPE(hmesh, "find-pick-intersec")(v,u,w,el,c)`

```
HMesh3D *hmesh
VEC3 v
double u,w
HELEMENT2D **el
double *c
```

This method determines for given screen-coordinates `u,w` the point on the mesh which is hit by these pick-coordinates. The world coordinates of this point are returned in `v`, the element on level `mesh->level_of_interest` which contains this point and its local-coordinates are returned in `el` and `c`. The only difference to the `Mesh3d`-version is the use of improved hierarchical searching.

`(HMESH3D *)GRAPE(hmesh, "hmesh-interactive-send")()`

```
HMesh3D *hmesh
```

This method sets up the project `uif-hm3`, which is described below.

#### 14.6.6 Affiliate Projects

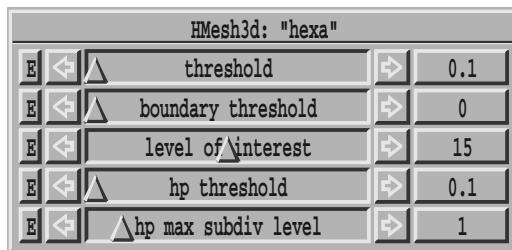


Figure 50: With the `HMesh3d` interface of the `uif-hm3` project certain parameters of the `HMESH3D` and `F_HDATA` structures can be adjusted interactively.

The project `uif-hm3` is the interactive user interfaces for `HMesh3d` instances. It allows editing `max_level_of_interest` and all `threshold` values of the `HMESH3D` and `F_HDATA` structure of the current object (cf. Figure 50). One can add them by hand, when needed, or they can be used in the user code with the function `g_project_use(name, batch)` or `g_project_add(name)`. For instance:

```
main ()  
{  
    ...  
    g_project_use ("uif-hm3", g_batch_mode);  
    ...  
}
```

## References

- [1] R. E. Bank and A. Weisser. Some a priori error estimators for elliptic partial differential equations. *Math. Comp.*, 44:283–301, 1985.
- [2] Bornemann, F. and Erdmann, B. and Kornhuber, R. Adaptive multilevel methods in three space dimensions. *Int. J. Numer. Methods Eng.*, 36, No.18:3187–3203, 1993.
- [3] H.-J. Bungartz, M. Griebel, D. Röschke, and C. Zenger. Pointwise convergence of the combination technique for the laplace equation. *East-West Journal of Numerical Mathematics*, 2(1):21–45, 1994.
- [4] A. Certain, J. Popović, T. DeRose, T. Duchamp, D. Salesin, and W. Stuetzle. Interactive multiresolution surface viewing. In *SIGGRAPH 96 Conference Proceedings*, pages 91–98, 1996.
- [5] P. Ciarlet and J. Lions. *Handbook of numerical analysis. Vol. V: Techniques of scientific computing*. Elsevier, 1997.
- [6] P. Cignoni, L. De Floriani, C. Montoni, E. Puppo, and R. Scopigno. Multiresolution modeling and visualization of volume data based on simplicial complexes. In *1994 Symposium on Volume Visualization*, pages 19–26, 1994.
- [7] D. S. Dyer. A dataflow toolkit for visualization. *Computer Graphics & Applications*, 10(4):60–69, 1990.
- [8] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH 95 Conference Proceedings*, pages 173–182, 1995.
- [9] B. Falcidieno, S. Orgolesu, C. Pizzi, A. Sanguineti, and M. Spagnuolo. High fidelity digital terrain modelling for the reconstruction of the antarctic sea floor. *Journal of Visualization and Computer Animation*, 7(3):177–188, 1996.
- [10] Froehlich, J. and Schneider, K. An adaptive wavelet-vaguelette algorithm for the solution of pdes. *J. Comput. Phys.*, 130, No.2:174–190, 1997.
- [11] M. Griebel. Eine kombinationsmethode für die lösung von dünnigitter-problemen auf multiprozessor-maschinen. In W. Bader, Rannacher, editor, *Numerische Algorithmen auf Transputer-Systemen*, pages 66–78. Teubner, 1992.
- [12] M. Griebel. Adaptive sparse grid multilevel methods for elliptic pdes based on finite differences. *Computing*, 1998. to appear.
- [13] M. H. Gross and R. G. Staadt. Fast multiresolution surface meshing. In *Proceedings of the Visualization*, pages 135–142, 1995.
- [14] R. Gross, C. Luerig, and T. Ertl. The multilevel finite element method for adaptive mesh optimization and visualization of volume data. In *Proceedings Visualization*, 1997.
- [15] R. B. Haber, B. Lucas, and N. Collins. A data model for scientific visualization with provisions for regular and irregular grids. In *Proc. IEEE Visualization*, 1991.
- [16] W. Hackbusch. *Multi-grid methods and applications*. Springer, Berlin/Heidelberg, 1985.

- [17] B. Hamann. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design*, 11:197–214, 1994.
- [18] Hemker, P.W. On the structure of an adaptive multi-level algorithm. *BIT*, 20:289–301, 1980.
- [19] N. Heusser and M. Rumpf. Efficient Visualization of Data on Sparse Grids. In *Proceedings VisMath Conference, Berlin*, 1997.
- [20] H. Hoppe. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings*, pages 99–108, 1996.
- [21] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *Transactions on Visualization and Computer Graphics*, 1(4):319–327, 1995.
- [22] R. Klein, G. Liebich, and W. Straßer. Mesh reduction with error control. In *Proceedings Visualization*, 1996.
- [23] D. Kroener. *Numerical schemes for conservation laws*. Series Advances in Numerical Mathematics. Wiley-Teubner, 1997.
- [24] U. Lang, R. Lang, and R. Röhle. Integration of visualization and scientific calculation in a software system. In *Proc. IEEE Visualization '91*, 1991.
- [25] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In T. W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 285–288, July 1991.
- [26] Y. Livnat, H. W. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm using the span space. *Transaction on Visualization and Computer Graphics*, 2(1):73–83, 1996.
- [27] B. Lucas and et. al. An architecture for a scientific visualization system. In *Proc. IEEE Visualization*, 92.
- [28] S. Muraki. Volume data and wavelet transform. *Computer Graphics and Applications*, 13(4):50–56, 1993.
- [29] R. Neubauer, M. Ohlberger, M. Rumpf, and R. Schwörer. Efficient Visualization of Large-Scale Data on Hierarchical Meshes. In W. Lefer and M. Grave, editors, *Visualization in Scientific Computing*. Springer, 1997.
- [30] H. Neunzert, W. Sack, and G. Koppenwallner. Development of a particle method for reactive flows in rarefied gases. In Hoffmann, Karl-Heinz (ed.) et al., *Mathematik: Schluesseltechnologie fuer die Zukunft. Verbundprojekte zwischen Universitaet und Industrie*. Springer, 1997.
- [31] M. Ohlberger and M. Rumpf. Hierarchical and Adaptive Visualization on Nested Grids. *Computing*, 59 (4):269–285, 1997.
- [32] M. Ohlberger and M. Rumpf. Adaptive projection methods in multiresolutional scientific visualization. Report 20, Sonderforschungsbereich 256, Bonn, 1998.

- [33] M. Rumpf, A. Schmidt, and K. Siebert. On a unified visualization approach for data from advanced numerical methods. In R. Scateni, J. Van Wijk, and P. Zanarini, editors, *Visualization in Scientific Computing*, pages 35–44. Springer, 1995.
- [34] M. Rumpf, A. Schmidt, and K. Siebert. Functions Describing Arbitrary Meshes. *Computer Graphics Forum*, 15 (2):129–141, 1996.
- [35] P. Schröder and W. Sweldens. Spherical wavelets: Efficiently representing functions on the sphere. In *SIGGRAPH 95 Conference Proceedings*, pages 161–172, 1995.
- [36] W. J. Schroeder, J. A. Zarge, and W. A. Lorensen. Decimation of triangle meshes. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, 1992.
- [37] Schwab, C. and Suri, M. The  $p$  and  $hp$  versions of the finite element method for problems with boundary layers. *Math. Comput.*, 65, No.216:1403–1429, 1996.
- [38] U. o. B. SFB 256. Grape manual. pages <http://www.iam.uni-bonn.de/main.html>.
- [39] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill. Octree-based decimation of marching cubes surfaces. In *Proceedings Visualization*. IEEE, 1996.
- [40] H.-W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *Proceedings Visualization*, 1996.
- [41] Stenberg, R. and Suri, M. Mixed  $hp$  finite element methods for problems in elasticity and Stokes flow. *Numer. Math.*, 72, No.3:367–389, 1996.
- [42] L. A. Treinish. Data structures and access software for scientific visualization. *Computer Graphics*, 25:104–118, 1991.
- [43] G. Turk. Re-tiling polygonal surfaces. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 55–64, July 1992.
- [44] C. Upson and et. al. The application visualization system: A computational environment for scientific visualization. *Computer Graphics & Applications*, 9(4):30–42, 1989.
- [45] J. P. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, 5, pages 57–62, 1990.
- [46] S. Yang and C. Cooke. Data compression based on the cubic B-spline wavelet with uniform two-scale relation. *Math. Comput. Modelling*, 23, No.7:73–88, 1996.
- [47] C. Zenger. Sparse grids. In W. Hackbusch, editor, *Parallel Algorithms for PDE, Proc. 6th GAMM Seminar Kiel, Notes on Numerical Fluid Mechanics*, volume 31, pages 241–251. Vieweg, 1991.