Functions defining Arbitrary Meshes

—

A Flexible Interface between Numerical Data and Visualization Routines

Martin Rumpf, Alfred Schmidt, Kunibert G. Siebert *

Institut für Angewandte Mathematik
Universität Freiburg
Hermann–Herder–Straße 10
D-79104 Freiburg i. Br., Germany

February 11, 1995

Abstract

Most of the rendering tools in scientific visualization are restricted to special data structures which differ substantially from the data formats used in numerical applications. Trying to close this gap, we present an interface between data from numerical methods on general types of grids — like cuboidal, prismatic, simplicial, parametric, mixed, or hierarchical meshes — and general visualization routines. It is based on a procedural approach managing a collection of arbitrary elements and a set of functions describing each element type. No mapping of (an in general enormous amount of) numerical data onto new data structures is necessary; a user may use his own data structures and only has to provide this small set of procedures and functions. The visualization tools will then use these routines to access (temporarily and locally) data of interest, like information about a single element. Compared with display routines on a specialized data structure, this general interface does not produce much cpu overhead.

AMS Classification: 65S05, 68P05, 68U05

Keywords: Visualization, arbitrary meshes, mixed elements, procedural access

*EMAIL: mart, alfred, kunibert@mathematik.uni-freiburg.de

1 Introduction

Visualization of numerical data is based on fast access to information about the underlying mesh (and some values given on that mesh) and display methods that work on this data, also called the "visualization model" [8].
There are a lot of numerical methods for the computational solution of (physical) problems which work on different types of underlying meshes. Finite Difference methods work on structured grids with a logical rectangular structure and implicitely defined adjacencies and nodes (see [23], e.g.). On the other hand, modern Finite Element and Finite Volume methods are built on locally refined unstructured grids with explicit element adjacency, mixed element types, and varying degrees of element parametrization and ansatz functions ([3], [4], [11], [20], [27]). A variety of data functions with different properties may be produced by the numerical methods. For example, data may be given by nodal values, a Lagrangian or hierarchical representation, or as a function piecewisely defined on parts of an element. Additionally, such data may be only locally continuous but discontinuous over element boundaries.
To understand qualitative and quantitative aspects of the calculated solutions, visualization is an essential tool. In addition, it supports development and debugging of new numerical algorithms, especially if they are based on complicated adaptive grid handling strategies. A desirable feature is the integration of the visualization routines into the numerical code; the visualization routines can use data of the numerical method, already available in main memory.
Thus, we are looking for a data interface between numerical methods and display methods, which meets the following requirements.

- The interface should be able to handle a lot of different types of meshes with data given on them (at least those mentioned above).
- The interface should not require more storage for the representation of the mesh and the numerical data on it than the underlying numerical method.
- The interface should include solely information about the mesh which is definitely needed by the display routines.
- To represent his own type of mesh, a user should be able to use the data structures he is accustomed to from his numerical method.

The gap between the user's numerical data formats and the structures usually used by rendering tools is a wellknown outstanding problem in scientific visualization [14, 24]. Most of the frequently used visualization software [9, 12, 15, 22, 25] works on prescribed data formats. A user has to convert his own data into such a format. Well known mesh formats handling a larger class of meshes is the UCD format implemented in AVS [1, 2] and the HDF format from NCSA [16] used in the GRASPARC data management system [6], e.g.
In search for conceptional new ideas, the methodology of fibre bundles has been introduced in [8]. The paper includes a taxonomy of geometries, functions on them, and the appropriate graphical operations. Related algorithms and data structures are discussed and applied to overlapping grids in [7], using a procedural handling of elementary geometry data (in the programming language Eiffel). [10] discusses a refinement of these ideas for scientific data on structured and unstructured grids; the geometry under consideration consists of "field elements", and classes of functions from Lagrangian function bases are discussed. Finally, subclasses of these arbitrary meshes, with less storage requirements, are inspected.
But there are still a lot of restrictions in these approaches; only a few element types are supported by the current software and the ansatz functions are limited to the most basic ones.

In our approach, we try to avoid restrictions on the ansatz functions and on the types of elements. A grid is defined as a procedurally linked list of mesh elements. Each single element is parametrized over a convex polyhedron with an arbitrary number of vertices and faces, perhaps varying from element to element. In this procedural approach, functions map from user's data to a single element; user's data structures are usually those from his numerical method, maybe even without an explicit mesh geometry. Given data on the mesh is accessed elementwise only. The next sections enligthen these points more precisely. Throughout the description of our approach, we will directly refer to the actual C structures used in our implementation to demonstrate the flexible features as clear as possible.

The paper is organized as follows: Sections 2 and 3 describe in more detail given numerical data and requirements of basic visualization methods, to define both sides around a general data interface. In section 4, we present our concept of such an interface, section 5 discusses the implementation of visualization methods, and section 6 presents some results. Implementational details and a description of the implemented visualization methods will be discussed in the appendix.

2 Numerical data revisited

Recent numerical methods deliver a large amount of data on a lot of different mesh types. The following list shows some of these types.

- Structured or unstructured meshes (from Finite Difference or Finite Element / Volume methods) with a variety of ansatz functions on them.
- Adaptive meshes consisting of a single or of mixed element types, e. g. simplicial, prismatic, rectangular or cuboidal structure [5, 20] or of more general kind (CSG modelling of surfaces with smooth corners naturally leads to triangles or pentagons, e. g.).
- Conforming or non–conforming meshes, where the neighbourhood of elements across an element side is not one–to–one (for example, locally refined rectangular meshes lead naturally to non–conforming meshes).
- Meshes with parametric (curved) elements of any of the above mentioned types, with globally constant or locally changing polynomial order of parametrization.
- Hierarchically structured meshes suitable for efficient numerical solvers (which give the possibility to choose a coarse hierarchy level for interactivity of time–consuming operations).

Finite Element / Finite Volume data is usually given by coefficients to ansatz functions with local support and can easily be evaluated in **local coordinates**. Therefore a function `f` on a specific element `e` of such a mesh can easily be evaluated by a call like `f(e,c)` where `c` is the local coordinate vector. For example, for piecewise linear data on triangles, the barycentric coordinates of a point in the triangle are equal to the values of the nodal basis functions. On the other hand, a piecewise polynomial interpolation of Finite Difference data, which are given only at the nodal points of a regular grid, can easily be evaluated using local coordinates as parameters.
As we want to handle 'general' elements, we use the following notion of local coordinates that belong to an element: For each element, we specify the dimension of the local coordinate system and the coordinates of all vertices of the element in that local system.

Usually, an n–dimensional element can be parametrized by an n–dimensional local coordinate system, but there are situations where it is more intuitive or appropriate to use a higher dimensional local system. Examples of this are:

- Usage of the full $(n+1)$–dimensional barycentric coordinate system on an n–dimensional simplex. If $P_1, \ldots, P_{n+1}$ are the vertices of the simplex, there is a unique representation of every point P of the simplex as $P = \lambda_1 P_1 + \ldots + \lambda_{n+1} P_{n+1}$ with $\lambda_i \geq 0$ and $\lambda_1 + \ldots + \lambda_{n+1} = 1$ [13].
- Smooth corners in a surface built out of rectangular patches may be naturally described as triangles with 3 local coordinates or pentagons with 5 local coordinates.

As a result, we have to permit local coordinate systems of arbitrary dimension. It is natural to assume that the underlying coordinate space is a standard Euclidian vector space. The following figure shows some examples of possible local coordinate systems for distinct elements. These elements are given as the following range of local coordinates:

$$
\begin{array}{ll}
\text{tetrahedron:} & \{(\lambda_0, \lambda_1, \lambda_2, \lambda_3) : \lambda_i \geq 0 \text{ and } \sum \lambda_i = 1\}, \\
\text{prism:} & \{(\lambda_0, \lambda_1, \lambda_2, h) : 0 \leq h \leq 1,\ \lambda_i \geq 0 \text{ and } \sum \lambda_i = 1\}, \\
\text{pyramid:} & \{(\mu_1, \mu_2, h) : 0 \leq h \leq 1 \text{ and } -(1-h) \leq \mu_i \leq (1-h)\}, \\
\text{cube:} & \{(\mu_1, \mu_2, \mu_3) : 0 \leq \mu_i \leq 1\}.
\end{array}
$$

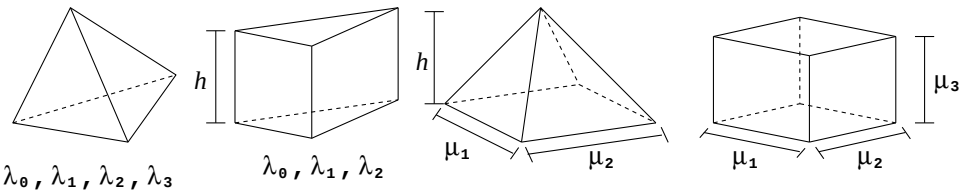

Fig. 1: Possible local coordinate systems of a tetrahedron, a prism, a pyramid and a cube

A general concept should allow to handle at least all of the above mentioned mesh types together with data from the underlying numerical methods. Again we should underline that not significantly more storage should be needed for the representation of the mesh than is used in the numerical methods. This applies especially to the case of structured grids, which can be represented by only a very small amount of data describing the mesh itself.

3 Essentials for visualization methods on arbitrary grids

Visualization methods can be formulated in a way that they need only a sequential access to elements of the mesh, either in a list–like order (when looping through all elements) or in a geometry– (neighbourhood–) related order (when tracing particles in a flow field, e. g.). None of the usual methods needs a direct access to an arbitrary element of the mesh. Most visualization methods for 2d or 3d meshes can be carried out by looping over all elements of the mesh and using only local information on each element.
Geometrical computations are done in local coordinates on a single element whereas world coordinates are needed for display tasks. Thus, all display methods need access to a transformation routine from local coordinates to world coordinates. Additionally, the inverse transformation from world to local coordinates is needed by some special methods; examples are data probes at points given in world coordinates and particle tracing methods (see section 5).

Some of the basic tasks which need only some local element information are:

- Display of the mesh or the boundary of the domain.
- Isoline and isosurface display (not the generation of a closed geometrical description, but only the display generation).
- Clipping of a volume against a plane or, more general, against an implicitly given surface like a ball.
- Visualizing specific data (as isolines, vectorfields etc.) on a clipping surface.

Additionally, there exist visualization tasks that demand more knowledge about the mesh, especially about the adjacency of elements:

- Display of element clusters (where the task may be to show a single element or a set of elements together with all adjacent elements), which needs information about all elements that are adjacent to one element.
- Particle tracing (generation of streamlines or streaklines), which again needs information about the adjacency of elements and additionally needs the transformation from local coordinates of one element to the local coordinates corresponding to an adjacent element.
- Search for an element which contains a given point in space (like the starting point for a particle trace), which can be done in convex domains by starting at an arbitrary element and jumping afterwards from one element to another, directed by local neighbourhood information.

To simplify and accelerate the visualization tasks, some methods are able to benefit from a *global* indexing of the element vertices and the elements. So, if possible, a global numbering of the vertices and the elements should accompany the mesh description, together with an upper bound for the maximal index. This kind of indexing does not mean that every index value is used, but only that every vertex resp. element refers to a unique index. A visualization routine that uses data values at the vertex positions (like an isoline routine) could use this global index to calculate values at the vertices only once, even if a node occurs as a vertex of several different elements (in a cuboidal mesh, each inner node occurs as a vertex of eight different cubes, e. g.).

4 Description of the mesh concept

The requirements from both numerical and visualization methods can be met by the data interface, which we describe in this section. It uses a procedurally linked list of elements, where the geometric type of each element may change from element to element. No fixed set of element types is prescribed, but a geometric description is used which is able to handle all types of elements which are parametrized over a convex polyhedron.
In the following sections, we describe the underlying concepts and structures (in C–notation) in a bottom–up way:

- element type description,
- element data,
- mesh data.

We present only the structures for threedimensional meshes. The corresponding structures for twodimensional meshes are straightforward and quite similar; the main difference is, that the description of 2D elements is a lot simpler.

Description of the geometric element type

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 some transformation routine.
For a threedimensional 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 outer 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 neighbourship of elements. Additionally, the transformation routine from world to local coordinates checks whether a given point is inside the element or not.

```
typedef struct
{
 int       number_of_vertices;    /* number of vertices                 */
 int       number_of_polygons;    /* number of faces                    */
 int       *polygon_length;       /* vertex counts of the faces         */
 int       **polygon_vertex;      /* face to vertex adjacency           */
 int       **polygon_neighbour;   /* face to face adjacency             */

 int       dimension_of_coord;    /* dim. of local coordinate system    */
 float     **coord;               /* local coordinates of the vertices  */
 int       parametric_degree;     /* parametric element if > 1          */
 int       (*world_to_coord)(const ELEMENT3D *, const float *, float *);
 void      (*coord_to_world)(const ELEMENT3D *, const float *, float *);
 int       (*check_inside)(const ELEMENT3D *, const float *);

 ELEMENT3D *(*neighbour)(ELEMENT3D *, const int, const int, float *, const float *);
 int       (*boundary)(const ELEMENT3D *, const int);
} ELEMENT3D_DESCRIPTION;
```

The arrays and procedures give following information:

`polygon_length[`j`]` is the count of vertices of the j–th face ($0 \leq j <$ `number_of_polygons`).

`polygon_vertex[`j`][`i`]` is the (local) index (with range $[0, \ldots,$ `number_of_vertices` $- 1]$) of the i–th vertex from the j–th face ($0 \leq j <$ `number_of_polygons`, $0 \leq i <$ `polygon_length[`j`]`).

`polygon_neighbour[`j`][`i`]` is the index of the adjacent face to the j–th face accross the edge from vertex `polygon_vertex[`j`][`i`]` to vertex `polygon_vertex[`j`][`$i+1$`]` ($0 \leq j <$ `number_of_polygons`, $0 \leq i <$ `polygon_length[`j`]`).

`coord[`k`][`i`]` is the i–th local coordinate of the k–th vertex from the polyhedron ($0 \leq i <$ `dimension_of_coord`, $0 \leq k <$ `number_of_vertices`).

`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)` 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 `NIL`, 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 `NIL`. The structure `element3d` may be overwritten by the routine.

`boundary(element3d, polygon)` returns 0, if `polygon` is an inner face of the mesh, or a nonzero integer, if `polygon` is part of the mesh's boundary.

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 the Figures 2–4, examples of vertex and face relationships for cubes, tetrahedra, and prisms are depicted. `check_inside()` is easily implemented using the local coordinates mentioned in section 1.

The user's data structures (from a numerical method, e.g.) may be completely different to the structures described above. In case of implicitely given Finite Difference meshes, the functions `neighbour` and `boundary` are implemented using simple index arithmetic (for a `C` implementation we refer to Appendix B). Unstructured grids usually provide such information explicitely.

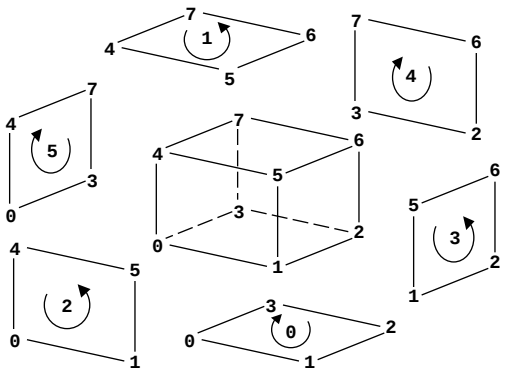

Fig. 2: Prototype of a cube

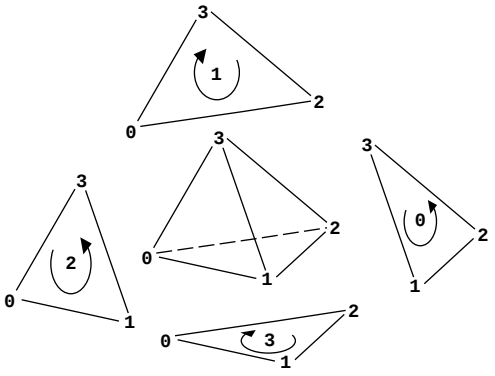

Fig. 3: Prototype of a tetrahedron

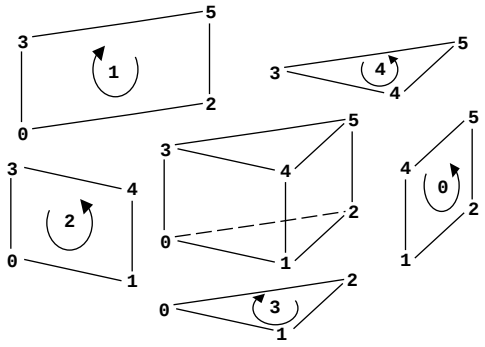

Fig. 4: Prototype of a prism

Element data

We use a procedural access to single elements in this mesh concept. The procedures return element data in a small structure. This structure for an element of the mesh consists mainly of a pointer to an element description, which contains all information described above, and a vector of pointers 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).

```
typedef struct
{
  MESH3D                *mesh;        /* this is an element of 'mesh'    */
  float                 **vertex;     /* world coordinates of vertices   */
  int                   *vindex;      /* indices of vertices             */
  int                   eindex;       /* index of the element            */
  ELEMENT3D_DESCRIPTION  *descr;      /* connectivity, ...               */

  int                   size_of_data; /* optional size of user data area */
  void                  *data;        /* optional pointer to user data   */
                                      /* of this element                 */
} ELEMENT3D;
```

Mesh

The mesh is just a collection of elements, as they are described in the previous section, together with an optional data function on the mesh.
We restrict the access to the 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),
ELEMENT *next_element(element),
```

which return the anchor to the list resp. the next element in the list.

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

```
ELEMENT *neighbour(element, polygon, flag, coord, xyz).
```

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

```
ELEMENT *copy_element(element)
```

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

```
free_element(element).
```

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 pointered 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 filling out vertex and index information for an element and refering 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 `MESH` 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, so 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 $[0, \ldots, \texttt{max\_eindex}-1]$ resp. $[0, \ldots, \texttt{max\_vindex}-1]$). If any one of the `max_?index` values is smaller or equal to zero, the elements do not give corresponding index information.

At last, the `MESH` structure contains an interface to data given on the mesh. Here we want to support a situation where different types of data at the same time are given 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 method 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 ansatz functions. To handle all these situations, we use a rather general interface for the functions on the mesh. Instead of describing functions in terms of a Langrangian basis, compare for instance [10], in our concept a user supplied function pointer evaluates any type of user function, for example also Hermitian or only piecewisely defined on several parts of a single element.
In detail the following entries are included in the `MESH` data structure:

— A function `f_info(el)` which gives information about the current data values on the mesh and on the current element. It returns the pointer to a structure that includes

  – a description of the current data values ("pressure", e.g.),
  – the current `dimension_of_value`,
  – the local `polynomial_degree` of data values on an element,
  – a flag `continuous_data` which is 1, if the current data is continuous between elements, and 0 otherwise.

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

— A data pointer `data` and the size of the underlying memory area `size_of_data`. This simplifies read and write operations and the simultaneous handling of multiple mesh3d data sets.

```
typedef struct
{
  ELEMENT3D     *(*first_element)(const MESH3D *);
  ELEMENT3D     *(*next_element)(ELEMENT3D *);
  ELEMENT3D     *(*copy_element)(const ELEMENT3D *);
  void          (*free_element)(ELEMENT3D *);

  int           max_number_of_vertices;  /* max. number of vertices of    */
                                         /* one single element            */
  int           max_eindex;              /* max. global element index     */
  int           max_vindex;              /* max. global vertex index      */

  int           max_dimension_of_coord;  /* max. dimension of the local   */
                                         /* coordinate system             */

  void          (*f)(const ELEMENT3D *, const int, const float[], float[]);
  F_INFO        *(*f_info)(const ELEMENT3D *);

  int           size_of_data;         /* optional size of user data       */
  void          *data;                /* optional pointer to user data    */
} MESH3D;

typedef struct
{
  char          *name;                /* textual data description         */
  int           dimension_of_value;   /* dimension of data values         */
  int           polynomial_degree;    /* local polynominal degree of data */
  int           continuous_data;      /* data globally continuous? (0/1)  */
} F_INFO;
```

5 A sketch of the basic visualization methods

Up to now we have discussed the fundamental structures of arbitrary meshes and especially the description of single elements. In the following we will briefly examine that they fulfill the requests of most of the visualization methods. Therefore we will focus on two significant rendering techniques, a clipping algorithm which visualizes a scalar function or a vectorfield

on the surface of intersection and the tracing of a single particle in a 3D vectorfield. This should point out that the set of functions describing a single element and the procedures managing a mesh by giving access to its elements form a natural and minimal set of routines to deal with arbitrary meshes.

Clipping: A variety of information may be displayed on the intersection of a volume with an implicitly given surface. The actual clipping of the elements is independent of the specific rendering style we will use later on. Let us suppose that the visible part of the volume should be that part where a given function `g` is non–positive. The intersection surface is then the zero level of `g` on the mesh. The following algorithm `find_clip()` runs over all elements searching for intersections. If successful on a certain element, it pushes a structure `clip_frame` describing the local intersection with the element into a rendering routine `render_clip()`. In there, the actual rendering takes place and for example the degree of refinement, a color shading or an isoline image of a scalar function, or the projection of a vectorfield is displayed. In addition, a routine `draw_boundary()` visualizes the remaining visible parts of the mesh boundary. To simplify the presentation, we did not make use of the possible indexing of elements and vertices.

```
void find_clip(const MESH3D *m, const float (*g)(), const void (*render_clip)())
{
  ELEMENT3D              *e;
  ELEMENT3D_DESCRIPTION  *descr;
  CLIP_FRAME             *clip_frame;
  int                    i,countp,countm; /* #vertices with g>=0 or g<0 */

  for (e = m->first_element(m); e != NIL; e = m->next_element(e)) {
    descr = e->descr;
    countp = countm = 0;
    for(i=0; i < descr->number_of_vertices; i++)
      if(g(e->point[i],var) >= 0) countp++;
      else countm++;
    if(countm) {
      for (i=0; i < descr->number_of_polygons; i++)
        if (descr->boundary(e,i)) draw_boundary(e,i,g);
      if(countp) {
        clip_frame = <local geometry of the clipping>;
        render_clip(clip_frame);
      }
    }
  }
}
```

Particle tracing: A fundamental technique to understand flow phenomena is to look at test sets, like single particles, curves, or surfaces moving in the 3D vectorfield and to record their traces. A procedure `trace()` following one particle for a period of time `T` starting at a position `x` in an element `e` is sketched below.
To keep the algorithm efficient on arbitrary grids and to avoid global searches, it is essential to use adjacency information. For the clarity of the algorithmical presentation we restrict ourselves here to a first order Euler scheme with step length `dt` to integrate the underlying ordinary differential equations.

```
void trace(float x[3], ELEMENT3D *e, const float dt, const float T)
{
  int                    i;
  float                  *c, v[3], t, s;
  ELEMENT3D_DESCRIPTION  *descr = e->descr;
  MESH3D                 m = e->mesh;

  <memory allocation for c[m->max_dimension_of_coord]>
  t=0.;
  descr->world_to_coord(e,x,c);
  while (t < T) {
    s = (t+dt < T) ? dt : T-t;
    m->f(e,-1,c,v);
    x = x + s * v;
    <direct rendering of the trace would take place here>
    descr->world_to_coord(e,x,c);
    while((i = check_inside(e,c)) != -1)
      e = descr->neighbour(e,i,FIRST_NEIGHBOUR,c,x);
  }
}
```

Here we do not discuss other existing rendering tools based on isosurfaces or on the extraction of specific sets of elements. Some additional examples can be found on the pictures pages.

6 Applications

Our implementation of the general interface for arbitrary meshes is embedded into the interactive graphical environment GRAPE developed at the SFB 256 at Bonn University [17, 18, 19, 26]. The staff at the Institut für Angewandte Mathematik at Freiburg University substantially contributes to its design and features.
A set of visualization methods is available from the authors. At the moment, there are still some restrictions. Isoline and isosurface rendering techniques are based on linear interpolation although the functions might be of higher order. Furthermore the integration techniques are not fully implemented. An outstanding task is also to include timedependent geometries and data.

We present the application of this implementation to two totally different numerical methods. The first one is a CFD–computation of vortex breakdown behind a cylindrical obstacle using a Finite Difference method (Figs. 5–10). This computation was done at the Aerodynamischen Institut of Prof. Krause at RWTH Aachen. We would like to thank for the opportunity to test our visualization tools on this data.
Figs. 5, 7, 8 show norm of the velocity, density and energy on a clipping plane. Fig. 6 gives a view on the vectorfield on two different planes. The color indicates the normal component of the projected vectors. Fig. 9 presents shrunken elements from a coarse version of the underlying 113x41x33 Finite Difference grid. Fig. 10 contains an isosurface of the norm of the velocity.
The second application is a Finite Element computation of the electric potential in a particle detector (Figs. 11-13). For the computation, we use a locally refined prismatic grid. Due to the presence of non–conforming nodes produced by the local refinement, we may either build a conforming closure of the grid involving pyramids and tetrahedra [20] or we may use a constrained approximation on the non–conforming purely prismatic grid [21].
In Fig. 11 we zoom into a coarse version of the conforming mesh containing the three different element types.

The last two figures show calculation on the non–conforming mesh using constrained approximation techniques. Fig. 12 shows isolines and the underlying grid on two clipping planes. In Fig. 13, we cut a step out of the computational domain and show the underlying grid.

Finally, we have to investigate the question what is the price in speed we have to pay for the flexibility we achieved by our general concept. The following table lists the rendering times (computation and graphical output on an SGI Extreme with R4400 Processor) of some basic visualization methods, applied to a mesh consisting of about 150,000 tetrahedra and 200,000 nodes from a Finite Element computation. Times are given from specialized display routines for tetrahedral meshes and from our general routines, producing equivalent results.

display method	rendering time (sec.)	
	tetrahedral grid code	general mesh code
clipping the volume against a ball	1.02	1.81
extracting a layer of elements	5.10	5.75
color shading on a clipping plane	1.46	2.23

Conclusions

We have described in this paper a concept how to attack visualization of data on arbitrary meshes. It overcomes some of the fundamental difficulties in data handling and closes a part of the gap between user's numerical data and rendering tools.
In comparison to dedicated software, a cpu overhead is produced by the general mesh concept. For our applications, this has turned out to be acceptable.

Picture page 1

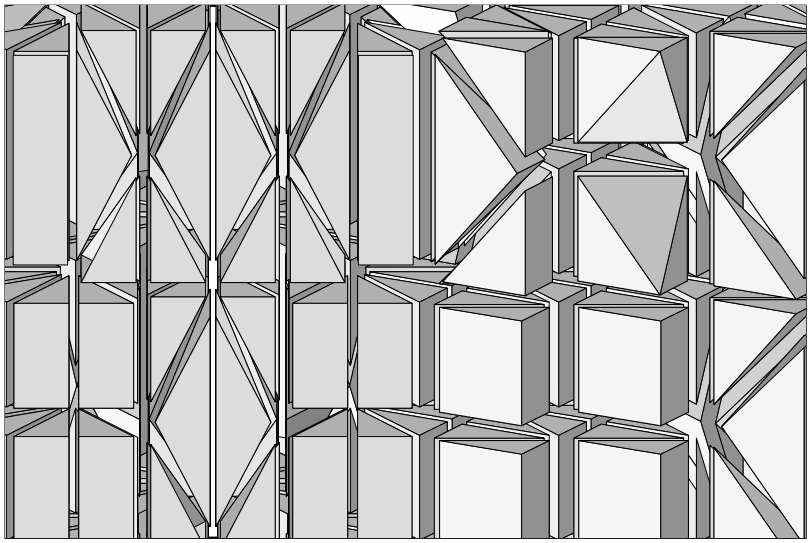

Fig. 11: A mixed type mesh with prisms, pyramids and tetrahedra

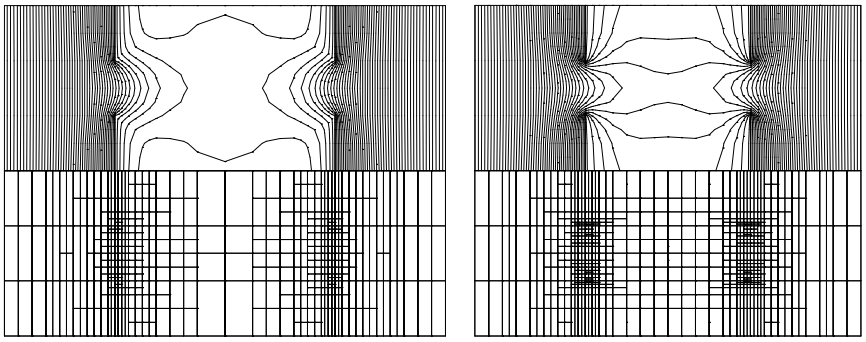

Fig. 12: Level lines and underlying grid of an electric potential

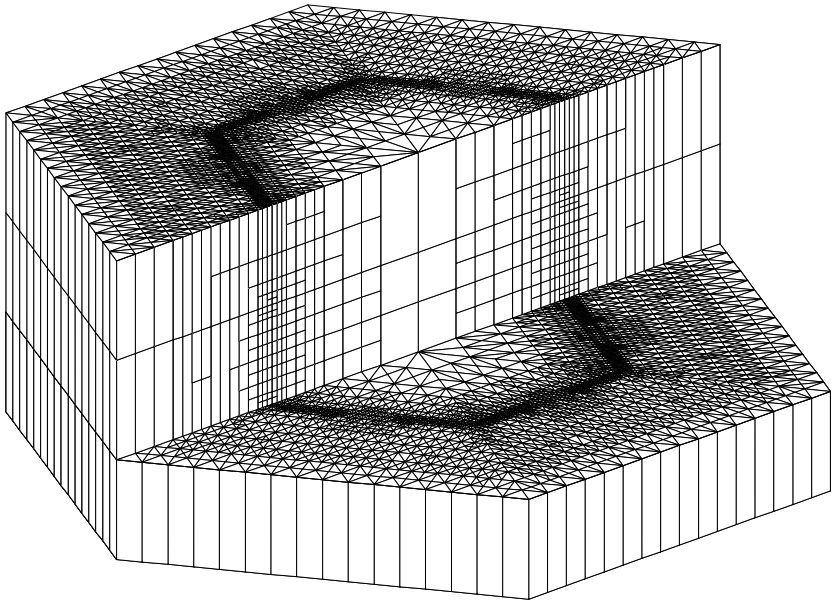

Fig. 13: Part of the pure prismatic mesh

References

[1] Advanced Visual Systems, Inc.: AVS developer's guide , Waltham, 1992

[2] Advanced Visual Systems, Inc.: AVS user's guide , Waltham, 1992

[3] Angermann, L.: Numerical solution of second–order elliptic equations on plane domains, Model. Math. Anal. Numer. 25, 169–191, 1991

[4] Demkowicz, L., Oden, J. T., Rachowicz, W., Hardy, O.: Toward a universal h–p adaptive finite element strategy, Part 1 – Part 3. Comp. Meth. Appl. Mech. Engrg. 77, 79–212, 1989

[5] Bänsch, E.: Local mesh refinement in 2 and 3 dimensions, IMPACT Comput. Sci. Engrg. 3, 181–191, 1991

[6] Brodlie, K.; et al.: GRASPARC – A problem solving environment integrating computation and visualization, Proc. IEEE Visualization 1993, 102–109

[7] Butler, D. M.; Bryson, S.: Vector–bundle classes form powerful tool for scientific visualization, Computer in Physics 6, 576–584, 1992

[8] Butler, D. M.; Pendley, M. H.: A visualization model based on the mathematics of fibre bundles, Computer in Physics 3, 45–51, 1989

[9] Dyer, D. S.: A dataflow toolkit for visualization, IEEE CG&A 10, No. 4, 60–69, 1990

[10] Haber, R. B.; Lucas, B.; Collins, N.: A data model for scientific visualization with provisions for regular and irregular grids, Proc. IEEE Visualization 1991

[11] Hackbusch, W.: On first and second order box schemes, Computing 41, 277–296, 1989

[12] IBM, Inc.: IBM AIX Visualization Data Explorer, user's guide, IBM Publication SC38–0081

[13] Kardestuncer, K.: Finite element handbook, McGraw-Hill, New York, 1987

[14] Lang, U.; Lang, R.; Rühle, R.: Integration of visualization and scientific calculation in a software system, Proc. IEEE Visualization, 1991

[15] Lucas, B.; et. al. : An architecture for a scientific visualization system, Proc. IEEE Visualization, 1992

[16] NCSA HDF specification manual, available via anonymous ftp from ftp.ncsa.uiuc.edu

[17] Rumpf, M.; Geiben, M.: Visualization of finite elements and tools for numerical analysis, Advances in Scientific Visualization, Eds. F. Post, A. H. Hin, Springer, 1993

[18] Rumpf, M.; Geiben, M.: Moving and tracing in timedependent vector fields on adaptive meshes, Report, SFB 256, Bonn, 1994

[19] Rumpf, M.; Schmidt, A.; et. al.: GRAPE graphics programming environment, Report 8, SFB 256, Bonn, 1990

[20] Siebert, K. G.: Local refinement of 3D–meshes consisting of prisms and conforming closure, IMPACT Comput. Sci. Engrg. 5, 271–284, 1993

[21] Siebert, K. G.: An a posteriori error estimator for anisotropic refinement, Preprint 313, SFB 256, Bonn, 1993 (to appear in Num. Math.)

[22] Silicon Graphics Computer Systems, Inc.: IRIS Explorer, Tech. Report BP–TR–1E–01, 1991

[23] Thompson, J. F.; Warsi, Z. U. A.; Mastin, C. W.: Numerical grid generation, North–Holland, 1985

[24] Treinish, L. A.: Data structures and access software for scientific visualization, Computer Graphics 25, 104–118, 1991

[25] Upson, C.; et. al.: The Application Visualization System: A computational environment for scientific visualization, IEEE CG&A 9, No. 4, 30–42, 1989

[26] Wierse, A.; Rumpf, M.: GRAPE, Eine objektorientierte Visualisierungs– und Numerikplattform. Informatik Forschung und Entwicklung 7, 145–151, 1992.

[27] Wierse, M.: Higher order upwind schemes on unstructured grids for the compressible euler equation in timedependent geometries in 3d, Dissertation, Freiburg, 1994

Appendix A (GRAPE Methods)

This section and the next one will recall and describe in detail the underlying class `Mesh3d`, the corresponding C structures and the basic visualization methods. We again emphazise that the work left to the user is to set up the interface dealing with his specific type of elements. The second part of this appendix will discuss concrete examples for parts of this interface. Nethertheless for Finite Difference applications two input formats are listed below. Furthermore tetrahedral meshes can be converted to `Mesh3d` instances.
This appendix should be seen as a reference manual.

Mesh3d

First we should recall for reference purposes `Mesh3d`, and collect the structures already introduced in the previous sections. The visualization system requires the following information:

- For each element type a description of the shape, functions to give information of adjacencies in the grid and functions to handle local and global coordinates: `ELEMENT3D_DESCRIPTION`
- For each element the whole geometric description: `ELEMENT3D`
- For the mesh a procedurally linked list with functions to create a first element and to run through the list: `MESH3D`

```
#define INSIDE -1
#define EXACT_NEIGHBOUR -1
#define FIRST_NEIGHBOUR 0
#define NEXT_NEIGHBOUR  1

typedef struct element3d_description  ELEMENT3D_DESCRIPTION;
typedef struct element3d              ELEMENT3D;
typedef struct mesh3d                 MESH3D;

struct element3d_description
{
 int       number_of_vertices;
 int       number_of_polygons;
 int       *polygon_length;
 int       **polygon_vertex;
 int       **polygon_neighbour;

 int       dimension_of_coord;
 float     **coord;
 int       parametric_degree;

 int       (*world_to_coord)(const ELEMENT3D *, const float *, float *);
 void      (*coord_to_world)(const ELEMENT3D *, const float *, float *);
 int       (*check_inside)(const ELEMENT3D *, const float *);

 ELEMENT3D *(*neighbour)(ELEMENT3D *, const int, const int, float *, const float *);
 int       (*boundary)(const ELEMENT3D *, const int);
};
```

```
struct element3d
{
  MESH3D                *mesh;
  float                 **vertex;
  int                   *vindex;
  int                   eindex;
  ELEMENT3D_DESCRIPTION *descr;
  int                   size_of_data;
  void                  *data;
};

typedef struct f_info
{
  char *name;
  int  dimension_of_value;
  int  polynomial_degree;
  int  continuous_data;
} F_INFO;

struct mesh3d
{ /* SYSTEM */
  CLASS         *class;
  char          *name;
  INSTANCE      *next;
  unsigned int  refcount;
  /* USER */
  ELEMENT3D     *(*first_element)(const MESH3D *);
  ELEMENT3D     *(*next_element)(ELEMENT3D *);
  ELEMENT3D     *(*copy_element)(const ELEMENT3D *);
  void          (*free_element)(ELEMENT3D *);
  int           max_number_of_vertices;
  int           max_eindex;
  int           max_vindex;
  int           max_dimension_of_coord;
  void          (*f)(const ELEMENT3D *, const int, const float[], float[]);
  F_INFO        *(*f_info)(const ELEMENT3D *);
  int           size_of_data;
  void          *data;
};
```

Methods on Mesh3d

GRAPE(mesh, "display")()
MESH3D *mesh

The `Mesh3d` instance `mesh` 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.

```
GRAPE(mesh, "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")()
    MESH3D *mesh
    float (*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 hypersurface 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 splitted 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 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 {float n_x, n_y, n_z, distance;} MPLANE_PARM;
extern float mplane(VEC3 x, MPLANE_PARM *var);

typedef struct {float center_x, center_y, center_z, radius;} MBALL_PARM;
extern float 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).

The interactive methods "clip-isoline-disp", "clip-deg-fine-disp", "clip-vect-disp" use these routines while calling the standard clipping method "clip".
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;
  float      *coord;
  float      (*f)();
  void       *var;
  void       *scal;
  int        flag ;
  int        dimension_of_value;
} CLIPM3D_PAR;
```

This structure describes the intersection of the hypersurface 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`$[0 \cdots n-1][0 \cdots 2]$ contains the global Euclidian 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 hypersurface.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).

GRAPE(mesh, "shrink")(p,f,fpar)
GRAPE(mesh, "shrink-disp")()
GRAPE(mesh, "shrink-focus-element-disp")()
GRAPE(mesh, "shrink-layer-disp")()
MESH3D *mesh
float p
int (*f)(ELEMENT3D *el,void *fpar)
void *fpar

The set of elements belonging to the Mesh3d instance is displayed on the standard device as a patch or grid model. They are separated from each other by a small gap of relative width `p`, this makes it possible to identify each element. There is a selection criterion for those elements which are of interest. The main method is called "shrink". Here a specific element is displayed if `f(ELEMENT3D *el, void *fpar)` is true otherwise it is skipped. There are several interactive versions with different predefined functions `f`.

- "shrink-disp" displays all elements.
- "shrink-layer-disp" displays all elements those center of mass of vertices lies in a layer surrounding a certain plane.
- "shrink-focus-element-disp" displays the neighbourhood of a certain elements. The index of this element and the depth of neighbourhood are accessible by integer rulers.

```
GRAPE(mesh, "level")(func,lev)
GRAPE(mesh, "level-disp")()
GRAPE(mesh, "sub-level-disp")()
GRAPE(mesh, "super-level-disp")()
    MESH3D *mesh
    void (*func)(ELEMENT3D *el, float *coord, float *val)
    float lev
```

The slices of those elements where the one-dimensional finite element function `func` has the level `lev` are calculated and displayed. The paramters of `func` are equal to those used for a one dimensional FEM function on a `Mesh3d` instance: `f(ELEMENT3D *el, float *coord, float *val)`. If `func` is NIL and `mesh` 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. The "super" and the "sub"-methods display the super or sub volumes of the function on the volume geometry.

Conversion Methods

The major purpose of the `Mesh3d` concept is to allow the access to data of various formats in a unique way. Therefore there is not one single format to read or write this data. It highly depends on the numerical method and the users data structures. Nevertheless some data formats will be supported. They will be discussed in the following.

```
(MESH3D *)GRAPE(tr3d, "convert-to-mesh")()
(MESH3D *)GRAPE(fe3d, "convert-to-mesh")()
GRAPE(scene, "convert-to-mesh-send")()
    TRIANG3D *tr3d
    FE3D *fe3d
```

A `Triang3d` or `Fe3d` object is referred to by a `Mesh3d` instance which is the returned value of the first two methods. The original triangulation and data are kept unchanged. The `Mesh3d` evaluation functions only refer to them. Later changes in `tr3d` or `fe3d` will therefore also effect the new `Mesh3d` instance. The methods "convert-to-mesh-send" on `Scene` replaces the object in the instance `scene` by a `Mesh3d` object refering to the previous object data.

```
(MESH3D *)GRAPE(Mesh3d, "read-ascii-fdiff")()
(MESH3D *)GRAPE(Mesh3d, "read-binary-fdiff")()
(MESH3D *)GRAPE(Mesh3d, "write-binary-fdiff")()
```

These three methods support the reading and writing of finite difference data on a possibly deformed cubic mesh. The values of several solution functions can be handled at the same time. A button in the interactive environment deals with the switching between them. An automatic compression of large data sets to a smaller space resolution is possible. A ruler effects the rate of discretizations points which will be taken into account in each coordinate direction.

Remark : The `coord_to_world` function works correctly if and only if the deformation of the standard reference cube onto the deformed cube has a constant normal direction in two directions and a varying one solely in the third direction. For arbitrary deformations the user

has to exchange this function by an appropriate one.

We will now list the input format for the read method dealing with the ASCII data type (the parentheses at the end of the lines contain comments) :

```
time

geo           (flag == 1 x_i y_i z_i are defined, == 0 not defined)
dim           (dimension of the solution vector)
parts         (number of segments in the solution vector,
              for example velocity, pressure, density, grad pressure ...)
part_0 name_0 (index to segment 0 of the function vector and its name)
.
.
.
part_(parts-1) name_(parts-1) (index of the last segment and its name)
n_x n_y n_z  (int number of steps in x,y,z direction)
             (n = n_x * n_y * n_z)
x_0 y_0 z_0 f_0[0]..f_0[dim-1] (x_i y_i z_i only if geo == 1)
     .                         (x is the slowest running index)
     .                         (z the fastest)
     .                         (point[i,j,k]=(x,y,z)[(i*y_n+j)*z_n+k])
     .                         (f_. solution vector on the points)
x_(n-1) y_(n-1) z_(n-1) f_(n-1)[0]..f_(n-1)[dim-1]
```

The following example illustrates this. A small 3D mesh is defined with a velocity and pressure defined on it.

```
0.0
1

4
2

0 velocity
3 pressure

3 3 3

0.0 0.0 0.0  1.0 2.0 1.0 0.012
0.0 0.0 0.5  1.0 2.0 1.0 0.012
0.0 0.0 1.0  1.2 1.6 0.5 0.018
0.0 0.5 0.0  1.2 1.6 0.5 0.018
0.0 0.5 0.5  0.9 2.2 0.7 0.008
0.0 0.5 1.0  1.2 1.6 0.5 0.018
0.0 1.0 0.0  1.0 2.0 1.0 0.012
0.0 1.0 0.5  1.2 1.6 0.5 0.018
0.0 1.0 1.0  0.9 2.2 0.7 0.008
0.5 0.0 0.0  1.2 1.6 0.5 0.018
0.5 0.0 0.5  1.0 2.0 1.0 0.012
0.5 0.0 1.0  1.2 1.6 0.5 0.018
0.5 0.5 0.0  0.9 2.2 0.7 0.008
0.5 0.5 0.5  1.0 2.0 1.0 0.012
0.5 0.5 1.0  0.9 2.2 0.7 0.008
0.5 1.0 0.0  1.2 1.6 0.5 0.018
0.5 1.0 0.5  1.0 2.0 1.0 0.012
0.5 1.0 1.0  0.9 2.2 0.7 0.008
1.0 0.0 0.0  1.0 2.0 1.0 0.012
1.0 0.0 0.5  0.9 2.2 0.7 0.008
1.0 0.0 1.0  1.2 1.6 0.5 0.018
1.0 0.5 0.0  0.9 2.2 0.7 0.008
1.0 0.5 0.5  1.0 2.0 1.0 0.012
```

```
1.0 0.5 1.0  0.9 2.2 0.7 0.008
1.0 1.0 0.0  1.0 2.0 1.0 0.012
1.0 1.0 0.5  1.0 2.0 1.0 0.012
1.0 1.0 1.0  1.0 2.0 1.0 0.012
```

If possible, the following binary format is preferable. It consists in replacing above all ASCII written integers and floats to binary written onces and the character strings by char arrays of fixed length 100. Furthermore, the block of function data information is separated from the coordinate arrays.

```
(float ) time

(int ) geo,dim,parts
(int ) part_0,(char[100] ) name_0
...
(int ) part_(parts-1),(char[100]) name_(parts-1)
(int ) n_x,n_y,n_z
(float ) x_0,y_0,z_0
...
(float ) x_(n-1),y_(n-1),z_(n-1)
(float ) f_0[0], .. f_0[dim-1]
...
(float ) f_(n-1)[0], .. f_(n-1)[dim-1]
```

Mesh2D

In the above discussion we always have considered the three dimensional case and argued that it is of more interest. Discussing the 2D structures is more or less straight forward. Here we present now the appropriate 2D structures including the class `Mesh2d`. The meaning of the structure entries are analogous to those already discussed in 3D.

```
typedef struct element2d_description  ELEMENT2D_DESCRIPTION;
typedef struct element2d              ELEMENT2D;
typedef struct mesh2d                 MESH2D;

struct element2d_description
{
 int       number_of_vertices;

 int       dimension_of_coord;
 float     **coord;
 int       parametric_degree;
 int       (*world_to_coord)(const ELEMENT2D *, const float *, float *);
 void      (*coord_to_world)(const ELEMENT2D *, const float *, float *);
 int       (*check_inside)(const ELEMENT2D *, const float *);

 ELEMENT2D *(*neighbour)(ELEMENT2D *, const int, const int, float *, const float *);
 int       (*boundary)(const ELEMENT2D *, const int);
};

struct element2d
{
  MESH2D                 *mesh;
  float                  **vertex;
```

```
  int                    *vindex;
  int                    eindex;
  ELEMENT2D_DESCRIPTION  *descr;
  int                    size_of_data;
  void                   *data;
};

struct mesh2d
{
  CLASS         *class;
  char          *name;
  INSTANCE      *next;
  unsigned int  refcount;

  ELEMENT2D     *(*first_element)(const MESH2D *);
  ELEMENT2D     *(*next_element)(ELEMENT2D *);
  ELEMENT2D     *(*copy_element)(const ELEMENT2D *);
  void          (*free_element)(ELEMENT2D *);

  int           max_number_of_vertices;
  int           max_eindex;
  int           max_vindex;
  int           max_dimension_of_coord;

  void          (*f)(const ELEMENT2D *, const int, const float[], float[]);
  F_INFO        *(*f_info)(const ELEMENT2D *);
  int           size_of_data;
  void          *data;
};
```

Appendix B, Examples for the implementation of element types

In the remainder we give some examples for the `ELEMENT3D_DESCRIPTION`. We point out one possible way how to define parts of the `ELEMENT3D_DESCRIPTION` for a cube, a prism, and a tetrahedron. For all elements we define `local_coord`, `polygon_length`, `polygon_vertex`, and `polygon_neighbour`. In the case of a cuboidal grid we define the main part of the function `neighbour()`. The second function `boundary()` uses the same ideas as `neighbour()` but is much easier to implement.

As a first example we start with one of the simplest mesh types and consider a set of equidistributed points given by a Finite Difference method. This should underline that beside arbitrary meshes containing various types of elements also the most basic grids can easily be handled by this concept. The whole grid is totally described by the constant mesh sizes `h_x`, `h_y`, `h_z` and the numbers of mesh points `n_x`, `n_y`, `n_z` in the x, y, and z–direction. Such information can be handled by the following

```
typedef
{
  int    n_x, n_y, n_z;     /* number of points in x, y, z-direction      */
  float  h_x, h_y, h_z;     /* distance of points in x, y, z-direction    */
  float  **f_data;          /* pointer to data matrix                     */
} USER_DATA;
```

This set of points describes a cuboidal grid. All cubes of the grid are of equal shape with diameters `h_x`, `h_y`, `h_z`. A single cube is totally described by the index tripel of one vertex, for example the vertex with smallest x, y and z coordinates.

```
typedef
{
  int i_x, i_y, i_z;        /* user's reference for the actual cube       */
} CUBE;
```

For a cube with index (`i_x, i_y, i_z`) the world coordinates of the reference vertex are simple (`i_x*h_x, i_y*h_y, i_z*h_z`). The world coordinates of the other vertices are calculated in a similar way. This is done by a procedure `fill_points(ELEMENT3D *el)`.
We use the following components of the `ELEMENT3D_DESCRIPTION` for one single cube (Fig. 2):

```
/* vertex indices of the polygons                                          */
static int c_v0[4] = {0,3,2,1}, c_v1[4] = {4,5,6,7}, c_v2[4] = {0,1,5,4};
static int c_v3[4] = {1,2,6,5}, c_v4[4] = {2,3,7,6}, c_v5[4] = {0,4,7,3};
/* polygon adjacencies                                                     */
static int c_p0[4] = {5,4,3,2}, c_p1[4] = {2,3,4,5}, c_p2[4] = {0,3,1,5};
static int c_p3[4] = {0,4,1,2}, c_p4[4] = {0,5,1,3}, c_p5[4] = {2,1,4,0};
/* local coordinates of the vertices                                       */
static float c_c0[3] = {0.,0.,0.}, c_c1[3] = {1.,0.,0.};
static float c_c2[3] = {1.,1.,0.}, c_c3[3] = {0.,1.,0.};
static float c_c4[3] = {0.,0.,1.}, c_c5[3] = {1.,0.,1.};
static float c_c6[3] = {1.,1.,1.}, c_c7[3] = {0.,1.,1.};

static float *cube_coord[8] = {c_c0,c_c1,c_c2,c_c3,c_c4,c_c5,c_c6,c_c7};
static int   cube_polygon_length[6] = {4,4,4,4,4,4};
static int   *cube_vertex[6] =        {c_v0,c_v1,c_v2,c_v3,c_v4,c_v5};
static int   *cube_next_polygon[6] =  {c_p0,c_p1,c_p2,c_p3,c_p4,c_p5};
```

The procedure which gives adjancency information on the grid can be implemented in the following way:

```
static ELEMENT3D  *cube_neighbour(ELEMENT3D *el, const int pn, const int flag,
                                  float *coord, const float *xyz)
{
  CUBE      *cd;
  USER_DATA *ud;

  cd = (CUBE*)el->data;            /* triple index of the acual element   */
  ud = (USER_DATA*)el->mesh->data; /* user data                           */

  if (flag == NEXT_NEIGBOUR)  return(NIL);

  switch(pn) {
    case 0 :
      if (cd->i_z == 0) return(NIL);
      cd->i_z--;
      fill_points(el);
      if (coord)  coord[2] += 1.0;
      return(el);
    case 1:
      if (cd->i_z == ud->n_z-2) return(NIL);
      cd->i_z++;
      fill_points(el);
      if (coord)  coord[2] -= 1.0;
      return(el);
    case 2:
      ...
  }
}
```

The description of a cube then simply is:

```
static ELEMENT3D_DESCRIPTION cube_descr =
{
  8, 6, cube_polygon_length, cube_vertex, cube_next_polygon,
  3, cube_coord, 1,
  cube_world_to_coord, cube_coord_to_world, cube_check_inside,
  cube_neighbour, cube_boundary
};
```

The basic functions of `MESH3D` are `first_element()` and `next_element()`. The first one has to allocate memory for one single element and for the triple index of the "user's element".

```
static ELEMENT3D *cube_first_element(const MESH3D *mesh)
{
  ELEMENT3D  *el;
  CUBE       *cd;

  if (mesh == NIL)  return(NIL);
  if ((el = (ELEMENT3D*)malloc(sizeof(ELEMENT3D))) == NIL)  return(NIL);

  el->mesh = (MESH3D *)mesh;
  el->vertex = (float**)malloc(8*sizeof(float*));
  el->vindex = (int*)malloc(8*sizeof(int));
  el->data = (void*)(cd = (CUBE*)malloc(sizeof(CUBE)));
  el->size_of_data = sizeof(CUBE);
  el->descr = &cube_descr;

  cd->i_x = cd->i_y = cd->i_z = 0;
  fill_points(el);
  return(el);
}

static ELEMENT3D *cube_next_element(ELEMENT3D *el)
{
  CUBE      *cd;
  USER_DATA *ud;
  if (el == NIL) return(NIL);
  cd = (CUBE*)el->data;
  ud = (USER_DATA*)el->mesh->data;

  if(++cd->i_x == ud->n_x-1) {
    cd->i_x = 0;
    if(++cd->i_y == ud->n_y-1) {
      cd->i_y = 0;
      if(++cd->i_z == ud->n_z-1) {
        cube_free_element(el);   /* el was last element of the grid      */
        return(NIL);
      }
    }
  }
  fill_points(el);
  return(el);
}
```

For the visualization of given data from a numerical method we have to implement a procedure `f()` with corresponding `f_info()`. Usually, Finite Difference methods only give values at the

nodes of the grid. Such data can be interpolated by a piecewise trilinear function. In the concept of local coordinates it is very easy to evaluate such a function. Let `u[0],...,u[7]` be those values at the vertices of one cube of the grid (for the sake of simplicity we assume that these are scalar values). The main part of the function `f(el, ind, coord, val)` would be

```
if (coord == NIL)
  *val = u[ind];
else {
  *val = (1-coord[2]) * ((1-coord[1])*((1-coord[0])*u[0]+coord[0]*u[1])+
                         coord[1]*((1-coord[0])*u[3]+coord[0]*u[2]));
  *val +=   coord[2]  * ((1-coord[1])*((1-coord[0])*u[4]+coord[0]*u[5])+
                         coord[1]*((1-coord[0])*u[7]+coord[0]*u[6]));
}
```

The values of `u[0],...,u[7]` can easily be obtained from the user's arrays (possibly stored at `el->mesh->data`) and some index arithmetic.

In the second example we give definitions for `local_coord`, `polygon_length`, `polygon_vertex`, and `polygon_neighbour` for tetrahedral and prismatic elements (see Figures 3 and 4). Since usually these elements occur in methods using unstructured grids, the adjacency connectivity of the elements can not be computed by indexing operations as described above. But for the Finite Element / Finite Volume methods itself the underlying user's data structure usually provides such information. Thus the functions `*neighbour()` and `*boundary()` can be implemented very easily from user's data.

```
/* vertex indices of the polygons for a prism and a tetrahedron           */
static int p_v0[4] = {1,2,5,4}, p_v1[4] = {0,3,5,2}, p_v2[4] = {0,1,4,3};
static int p_v3[3] = {0,2,1},   p_v4[3] = {3,4,5};
static int t_v0[3] = {1,2,3},   t_v1[3] = {2,0,3};
static int t_v2[3] = {0,1,3},   t_v3[3] = {2,1,0};

/* polygon adjacencies  for a prism and a tetrahedron                     */
static int p_p0[4] = {3,1,4,2}, p_p1[4] = {2,4,0,3}, p_p2[4] = {3,0,4,1};
static int p_p3[3] = {1,0,2},   p_p4[3] = {2,0,1};
static int t_p0[3] = {3,1,2},   t_p1[3] = {3,2,0};
static int t_p2[3] = {3,0,1},   t_p3[3] = {0,2,1};

/* local coordinates of the vertices for a prism and a tetrahedron        */
static float p_c0[4] = {1.,0.,0.,0.}, p_c1[4] = {0.,1.,0.,0.};
static float p_c2[4] = {0.,0.,1.,0.}, p_c3[4] = {1.,0.,0.,1.};
static float p_c4[4] = {0.,1.,0.,1.}, p_c5[4] = {0.,0.,1.,1.};
static float t_c0[4] = {1.,0.,0.,0.}, t_c1[4] = {0.,1.,0.,0.};
static float t_c2[4] = {0.,0.,1.,0.}, t_c3[4] = {0.,0.,0.,1.};

static int   prism_polygon_length[5] = {4,4,4,3,3};
static int   *prism_vertex[5] =       {p_v0,p_v1,p_v2,p_v3,p_v4};
static int   *prism_next_polygon[5] = {p_p0,p_p1,p_p2,p_p3,p_p4,};
static float *prism_coord[6] =        {p_c0,p_c1,p_c2,p_c3,p_c4,p_c5};

static int   tetra_polygon_length[4] = {3, 3, 3, 3};
static int   *tetra_vertex[4] =       {t_v0,t_v1,t_v2,t_v3};
static int   *tetra_next_polygon[4] = {t_p0,t_p1,t_p2,t_p3};
static float *tetra_coord[4] =        {t_c0,t_c1,t_c2,t_c3};
```

The `ELEMENT3D_DESCRIPTION` for a prism and a tetrahedron then would be

```
static ELEMENT3D_DESCRIPTION prism_descr =
{
  6, 5, prism_polygon_length, prism_vertex, prism_next_polygon,
  4, prism_coord, 1,
  prism_world_to_coord, prism_coord_to_world, prism_check_inside,
  prism_neighbour, prism_boundary
};
static ELEMENT3D_DESCRIPTION tetra_descr =
{
  4, 4, tetra_polygon_length, tetra_vertex, tetra_next_polygon,
  4, tetra_coord, 1,
  tetra_world_to_coord, tetra_coord_to_world, tetra_check_inside,
  tetra_neighbour, tetra_boundary
};
```