A Shapes based geometric modeler for Mesh++, API description

Monica Camba (mcamba@crs4.it) Piero Pili (piero@crs4.it) Gianluigi Zanetti (zag@crs4.it)

Document id: `VIVA-1998-7-1`
Revision: 0.0.1
Document type: `TECHREP`
Origin: `CRS4`

CRS4
Centro di Ricerca, Sviluppo e Studi Superiori in Sardegna
VI Strada OVEST Z.I. Macchiareddu
09010 UTA (CA - Italy)

Document History

Event	Date	Actor	Reference
CREATION	3/7/98	mcamba@crs4.it	

Contents

1 Introduction

One of the purposes of the **ViVa Project** is the reconstruction of geometrical models from a suitable set of data provided by non invasive medical analyses like Computed Tomography (CT) or Magnetic Resonance Imaging (MRI). To do this, we first segment the vessel geometry from the data sets using techniques such as active snakes, and then reconstruct geometrical models using the *XOX* **SHAPES** geometric modeler, and in particular its **MicroTopology** features. In order to perform blood flow simulation on these models, we need to mesh them, and so we use the **Mesh++** grid generator to process the models and create a two-dimensional grid for the surfaces and a three-dimensional one for the vessel lumen.

Mesh++ has his own geometric B-rep modeler inside, and so it is necessary to interface the **SHAPES** modeler to the native **Mesh++** modeler through an appropriate routines set. This document describes the resulting Application Procedure Interface (API).

The grid generator and **SHAPES** communicate through files containing a given amount of geometrical and topological information that **Mesh++** can handle. The topology of the geometric model is described by marking each of its elements (vertices, edges, faces) by an integer index. The grid generator reads the indices directly from the topology file and handles all the geometric objects tagged by the indices only through the interface facilities. The application interface manages the correspondence between indices and geometrical entities (loading a curve or a surface) and all the evaluations performed by the grid generator on geometries: it allows to compute the value of a function and its first and second derivatives in a given point of the parameter space, and the projection of a point over a surface or a curve.

Since the mesher was previously interfaced with another geometric modeler, the same interface has been maintained; the API routines have been implemented again using **SHAPES** facilities, in such a way that the **Mesh++** code remained the same: the memory areas passed to these functions are now used to allocate **SHAPES** objects handled always by the API routines and never directly by the mesher.

Besides, while in the beginning all the topological information about the model were not given by the geometric modeler, the topology is now extracted directly by **SHAPES** through suitable routines implemented in the **mesh++Topology** program.

If we want the mesher deliver a conformal mesh over two adjacent faces, they have to share a single edge, or a set of consecutive ones. In other words, the topological description does not allow two edges overlapping completely or partially. To avoid this event, we choose to modify the model description given to the mesher instead of the model itself in case of exactly overlapping edges.

2 API Purposes

The mesher has to mesh geometric objects that are given by **SHAPES**, and it accomplishes this task by performing evaluations on these **SHAPES** objects, so the API needs a suitable library (**gmesh**) containing all the routines able to carry out these tasks.

Besides, it is necessary to write the topology file, that translates the topology of the model in a format the mesher can treat; this task is executed by running a specialized program (**mesh++Topology**), while the former ones must be performed during the mesher execution. In fact, it is important to say that the **mesh++Topology** program extracts the topology of the model directly from the **SHAPES** objects, and that this topology does not match always the topology written out in the mesher format.

There is another set of facilities that are used by both the **mesh++Topology** and the **Mesh++** program, that allow to load the topology directly from the **SHAPES** geometric objects.

3 Modules organization

The **mesh++Topology** program is able to:

- load a geometric model by a **SHAPES** input file;
- draw out from the model the topological entities list;
- initialize suitable structures describing the model;
- find out the adjacency relationships between topological objects;
- write in a output file the above information;
- assign to the model default boundary conditions;
- map the default boundary conditions to other ones read by file;
- dump out, for each topological entity, a file containing the mesh information in case of **SHAPES** web geometries.

The **gmesh** library implements the following functionalities:

- to load the geometric model from a **SHAPES** file;
- to load the topology from a suitable file;
- to perform evaluations on the geometric entities listed in the topology file.

These tasks are so divided into the program modules:

- **loadModel.c**: initializing and ending the **SHAPES** work session, loading a **SHAPES** geometric model and processing its topology;
- **attr.c**: managing the **SHAPES** attributes describing the topological model;
- **metric.c**: distances and lengths computing;
- **ais_grids.c**: performing evaluations on the model;
- **mesh++Topology.c**: extracting the topology from the model and writing it in the output file.

The **gmesh** library is composed by the **loadModel**, **attr**, **metric** and **ais_grids** object files, while the **mesh++Topology** program links only the former three.

3.1 The gmesh library

The **gmesh** library is the actual interface between **SHAPES** and **Mesh++**; its functionalities are essentially divided in two blocks:

- loading a geometric model by a **SHAPES** input file and its topological descpription by a mesh++.input file;
- evaluations performing on the geometrical objects listed in the topology file.

The first goal is executed by the **loadModel** module, that uses the **attr** module functionalities to handle the attributes, while the evaluation functions compose the **ais_grids** module, using also the **metric** module facilities. The evaluation functions perform the following tasks:

- loading a curve or a surface from its index read by the mesh++.input file;
- evaluation of the coordinates in the modeling space of a point corresponding to given values of the parameters;
- evaluation of the first and second [1] derivatives of a curve or a surface in a given point of its parameter space;
- projection of a point over a curve or a surface.

Since the mesher assumes that all the geometric objects are parameterized in a suitable set of canonical intervals not corresponding to the actual **SHAPES** parameterizations, all the above functions contain a remapping of all the values passed from and to the mesher as input and output.

4 The geometric model

The **mesh++Topology** module has to give to **Mesh++** the set of data needed, that is a **SHAPES** file containing the geometry of the object to mesh and a file containing its topology.

This file is written in a suitable format that the mesher can read, described in [1]. Actually, the **SHAPES** file containing the geometry of the model contains also its topological information, but in a format not directly available to the mesher.

The **mesh++Topology** program does not create itself the geometrical model. This is a responsability of other programs, (**tube_builder**, **buildAnastomosis**, **buildBifurcation**), whose aim is in fact the reconstruction of the geometric model.

Moreover, the mesher cannot guarantee the mesh boundary conformity on a couple of faces sharing two different overlapping edges, so the **mesh++Topology** program has to hide this eventuality to **mesh++**.

5 The hiding algorithm

It is an opportune remark to say that the mesher does not guarantee the boundary conformity of the mesh if the model has two overlapping edges. In order to resolve this problem the **mesh++Topology** program, during the search for the vertices and edges of the geometry, performs a check on them and writes on the output file only one of two overlapping vertices or edges.

This check is aimed to "hide" to the mesher one of the vertices or edges of each couple of overlapping ones; the hiding of these entities happens while processing the geometry searching for its topological elements. It assumes that the correspondence between overlapping edges is one-to-one and that overlapping edges coincides exactly (that is, the set of their points in the modeling space is the same).

5.1 Attributes

In order to dump only one of a set of overlapping topological entities, we are going to mark one of them as the *master* and all the other as *slaves*. Each *master* element has an incremental integer index and all its *slaves* have the same index too. So, the API contains two **SHAPES** attribute specifiers to keep track of these relationships:

- an attribute specifier *masterType* with two attributes:
  - *master*

[1] Actually, **SHAPES** does not support evaluation of second derivatives of a surface with respect to two different parameters

  - *slave*

- an attribute specifier *indexType*.

  For each *master* vertex or edge it is created a new *indexType* attribute with a progressive index stored in.

5.2 Topological elements dumping

The **mesh++Topology** program processes all the topological elements of a given model, marks them as *master* or *slave*, attaches to each an *index* attribute, and finally dumps them to the topology file. It processes first the vertices, then the edges, and after this is ready to write the topology file. In this procedure, the vertices and edges are dumped to the topology file differently with respect to its parts: since the topology file is divided into *geometry keywords* and *topology keywords*, they can be treated in distinct ways. The *geometry keywords* part needs the elements to be listed in incremental order, that is exactly the order matching a global list of elements. This section allows to dump overlapping elements (the first list is there for future extensions; the current version of the mesher only uses the second list); it is a task of the second section to make sure that only one of a set of coincident elements will be used to perform evaluations. This is performed dumping the elements in the *topology keywords* part by their *index* attribute and not by their list index, since the *slave*s are marked with the same *index* attribute as their *master*.

5.3 Vertices marking

The list of all vertices **SHAPES** xGEOMs is processed, keeping track of already processed vertices (APV). All the APV are marked with the *master* attribute and with an incremental *index* attribute. Once a new vertex is processed, it is searched for all the APV whether it coincides with one of them. The check is performed with respect to a local tolerance to be passed to the program as an argument; if the check returns true, the processed vertex is marked as *slave* and it is attached to it the *index* attribute of its *master*, otherwise it is marked as *master* and receives an incremental *index* attribute. In this case it is of course included in the APV list.

5.4 Edges marking

The list of all edges **SHAPES** xGEOMs is first of all processed to give each of them a hashing key so defined: let e be an edge and v_1 and v_2 the *index* attributes of its vertices. The key to be associated to e is computed as:

$$key(e) = min(v_1, v_2)hashShift + max(v_1, v_2)$$

where $hashShift$ is an integer constant. All these keys are stored in a suitable array (*allKeys*). Then all the edges are processed in this way: it is created an already processed edges (APE) array, that will contain the position corresponding to an edge in the *allKeys* array. Once an edge is inserted in the APE array, it is marked as *master* and it is associated to it an incremental *index* attribute. If e is the current edge to be processed, it is compared with all the APE: if it has the same hashing key of an edge E in APE, it is executed a check on the point of e corresponding to the middle value of its parameter range: if this point is contained in E, it is clear that e is a *slave* of E, and so it is marked as *slave* and it is attached to it the E *index* attribute. If e has not the same key of any of the APE or if the e "middle parameter" point it is not contained in E, e is inserted in the APE array.

6 Boundary Conditions

The **mesh++Topology** program is able to detect whether the input geometry has boundary conditions, given as **SHAPES** attributes attached to its 1d boundaries. From these attributes, the program can compute default boundary conditions for all the other geometric entities. Otherwise, if the user specified an input labels file, the program processes it and maps the default values to the ones given by the file.

The input labels file has to contain couples of values like:
defaultValue newValue

The boundary conditions are then dumped into the mesh++.input file.

7 Mesh information

If the input geometry is a **SHAPES** web, **mesh++Topology** can dump a mesh data file for each topological element. These files are used by **Nast++** (see **Nast++** documentation) to perform projections aimed to the grid refinement.

The output file names include a keyword (point,line or surface) and an index that identifies the object in the mesh++.input topology file.

Every file contains:

- the file name capitalized keyword (POINT, LINE, SURFACE) and the space dimension (first row);
- the number of the web vertices and the number of the web simplices (second row);
- the point space coordinate and parameter values (for each web vertex);
- the index of each vertex w.r.t. the previous list (for each web simplex).

8 Input File

The input file has to be a **SHAPES** file containing a single xGEOM or a list of xGEOMs; in the latter case (and only in this case) the name of the file must contain the string ".gms". Every geometry in the file is expected to be a two dimensional geometry. If it is a xGROUP, it must have one single child.

9 Output File

The output file describes the topology of the model in the format read by the mesher [1]. This topology does not match exactly the topology of the input geometry, because the mesher is not able to treat it; so the topology output file contains the topology obtained from the **SHAPES** model filtered by the hiding algorithm.

10 Syntax

The syntax required to run the mesh++Topology program is:

```
mesh++Topology --infile=<SHAPESFILE> --outfile=<FILE.INPUT>
               --tol=<N> [--labels=<LABELSFILE>] [--mesh] [--help]
```

where:

- **SHAPESFILE** is a **SHAPES** file containing the geometrical model;
- **FILE.INPUT** is the target file for the topology;

- **N** is the power of 10 to be used as tolerance for coincident vertices;
- **LABELSFILE** contains the mapping of the boundary conditions values;
- `--mesh`

  outputs the mesh information for each topological entity;
- `--help`

  prints usage

10.1 Usage examples

- ```
  mesh++Topology --infile=cylinder.gm --outfile=cylinder.input
                 --tol=-3
  ```

  It reads the geometrical model from the *cylinder.input* file.

  It dumps to *cylinder.input* topological information and (if any) default boundary conditions.

  It uses 10^{-3} as local tolerance.
- ```
  mesh++Topology --infile=cylinder.gm --outfile=cylinder.input
                 --tol=-3 --labelsfile=myLabelsFile
  ```

  It dumps to *cylinder.input* topological information and (if any) boundary conditions read from *myLabelsFile*.
- ```
  mesh++Topology --infile=cylinder.gm --outfile=cylinder.input
                 --tol=-3 --mesh
  ```

  It produces a mesh data file for each topological element listed in the *cylinder.input* file.
  ```

11 Routines description

```
/*****************************************
evps1d

Purpose : It computes position vector and derivatives on a
          parametric curve in the tridimensional space.
Synopsis :
  int                   /* R  : non-zero on error */
  evps1d(
         float *xl,     /* I  : address of the geom to be evaluated */
         float *u,      /* I  : parametric coordinate local to the arc */
         float *xp,     /* O  : position vector r evaluated in u */
         float *xd,     /* O  : first derivative r,u */
         float *xdd,    /* O  : 2nd derivative r,uu */
         float *xddd,   /* O  : third derivative r,uuu */
         float *sz,     /* O  : first derivative modulus ||r,u|| */
         int   *in      /* I  : mode for evaluations */
  )
Description : This function performs evaluations over a curve.
        xl is the address of the geometry to be queried.
           It has to be a 1D xCELL and not a xGROUP.
        u contains the value of the parameter defining the
          evaluation.
          The parameter value belongs to a canonical range defined
          by the T0 and T1 constants in the file shInterface.h,
          and the function evps1d is able to scale it in the SHAPES
          parameter range of the geom representing the curve.
        xp is the place to store the position vector of the curve
           evaluated in the value u of the parameter.
        xd is the place to store the first derivative of the curve
           evaluated in the value u of the parameter.
        xdd is the place to store the secund derivative of the curve
            evaluated in the value u of the parameter.
        xddd is the place to store the third derivative of the curve
             evaluated in the value u of the parameter.
             Now it is not used.
        sz on output will contains the length of the first
           derivative.
        in is the mode used to perform the evaluations:
            0: position vector, 1st, 2nd and 3rd derivative;
            1: only position vector;
            2: position vector and 1st derivative;
           -1: only 1st derivative;
               The function can't currently evaluate third
```

```
             derivatives, because SHAPES doesn't support it.
             The function returns SH_OK if successful and SH_ERROR
             otherwise.


/******************************************
evsurg

Purpose : Evaluate point position and derivatives corresponding to
          the parametric coordinates u,v on a surface.
Synopsis :
  int                    /* R  : non-zero on error */
  evsurg(
         float *aps,      /* I  : address of the geom to be queried */
         float *u,        /* I  : local parametric coordinates */
         float *v,        /* I  : local parametric coordinates */
         int   *ndu,      /* I  : dummy argument */
         int   *ndv,      /* I  : dummy argument */
         float  r[3],     /* O  : position vector */
         float  ru[3],    /* O  :  1st derivatives r,u */
         float  rv[3],    /* O  :  1st derivatives r,v */
         float  ruv[3],   /* O  :  2nd derivatives r,uv */
         float  ruu[3],   /* O  :  2nd derivatives r,uu */
         float  rvv[3],   /* O  :  2nd derivatives r,vv */
         int   *ind       /* I  : mode for evaluations */
  )
Description : This function performs evaluations over a surface.
        aps is the address of the geometry to be queried.
            It has to be a 2D xCELL and not a xGROUP.
        u contains the value of the first parameter defining the
          evaluation.
        v contains the value of the secund parameter defining the
          evaluation.
            The parameter values belong to a canonical range defined
            by the T0 and T1 constants in the file shInterface.h,
            and the function is able to scale it in the SHAPES
            parameter range of the geom representing the surface.
        ndu is a dummy argument.
        ndv is a dummy argument.
        r is the place to store the position vector of the surface
          evaluated in the values u and v of its parameters.
        ru is the place to store the first derivative w.r.t. u of
           the surface evaluated in the values u and v of its parameters.
        rv is the place to store the first derivative w.r.t. v of
           the surface evaluated in the values u and v of its parameters.
```

```
        ruu is the place to store the secund derivative w.r.t. u of
            the surface evaluated in the values u and v of its parameters.
        ruv is the place to store the secund derivative w.r.t. u and v
            of the surface evaluated in the values u and v of its
            parameters.
        rvv is the place to store the secund derivative w.r.t. v of
            the surface evaluated in the values u and v of its parameters.
        xddd is the place to store the third derivative of the curve
             evaluated in the value u of the parameter.
             Now it is not used.
        sz on output will contains the length of the first
           derivative.
        ind is the mode used to perform the evaluations:
            -1: only ru and rv
             0: all
             1: only position vector r
             2: only r and 1st derivs r,u and r,v
               The function currently can't evaluate secund
               derivatives with respect to u and v, because SHAPES
               doesn't support it.
               The function returns SH_OK if successful and SH_ERROR
               otherwise.


/*****************************************
xlenst

Purpose : Compute the length of a segment of parametric
          curve between the points corresponding to two values
          of its parameter.
Synopsys :
  float                  /* R  : length of arc between u1 and u2 */
  xlenst(
         float *xln1d,   /* I  : address of the geom to be queried */
         float *u1,      /* I  : local parametric coordinate of first point */
         float *u2,      /* I  : local parametric coordinate of second point */
         float *eps,     /* I  : dummy argument */
         float *prec,    /* I  : dummy argument */
         int   *in,      /* I  : dummy argument */
         int   *ierr     /* I  : dummy argument */
  )
Description : This function evaluate the arc length over a curve.
        xln1d is the address of the geometry to be queried.
              It has to be a 1D xCELL and not a xGROUP.
        u1 contains the first value of the parameter defining the
```

```
      evaluation.
   u2 contains the secund value of the parameter defining the
      evaluation.
         The parameter value belongs to a canonical range defined
         by the T0 and T1 constants in the file shInterface.h,
         and the function is able to scale it in the SHAPES
         parameter range of the geom representing the curve.
   eps is a dummy argument.
   prec is a dummy argument.
   in is a dummy argument.
   ierr is a dummy argument.
          The function returns the length of the arc between
          u1 and u2.


/******************************************
locps4

Purpose : Find the closest vertex from a point to a surface.
Synopsis :
  int                      /* R  : non-zero on error */
  locps4(
         int   *n,         /* I  : dummy argument */
         int   *m,         /* I  : dummy argument */
         int   *keypa,     /* I  : dummy argument */
         float *aps,       /* I  : address of the geometry to be queried */
         float *xp,        /* I  : coordinates of given point */
         float *ug,        /* O  : u parameter of the closest point */
         float *vg,        /* O  : v parameter of the closest point */
         float *u10,       /* I  : dummy argument */
         float *u20,       /* I  : dummy argument */
         float *v10,       /* I  : dummy argument */
         float *v20,       /* I  : dummy argument */
         float *r,         /* O  : found minimum r=r(ug,vg) */
         float *z,         /* O  : r(ug,vg) - xp */
         float *dist,      /* O  : distance ||r(ug,vg) - xp|| */
         int   *imes,      /* I  : dummy argument */
         int   *iter,      /* I  : dummy argument */
         float *xinit[],   /* I  : dummy argument */
         float *utl,       /* I  : dummy argument */
         float *distl,     /* I  : dummy argument */
         float *ftl,       /* I  : dummy argument */
         float *zerom      /* I  : dummy argument */
  )
Description : This routine finds the minimum of the function
```

```
      DISTPS = || r - xp || where xp is a point on the 3d
      space and r a point on a surface.
      The search is limited to a prescribed interval.
n is a dummy argument.
m is a dummy argument.
keypa is a dummy argument.
aps is the address of the geometry to be queried.
    It has to be a 2D xCELL and not a xGROUP.
xp is the address of the array containing the coordinates
   in the modeling space of the given point.
ug is the value of the u parameter corresponding to the
   point of the surface closer to the given point xp.
vg is the value of the v parameter corresponding to the
   point of the surface closer to the given point xp.
      The values ug and vg belong to the range defined
      by the constants T0 and T1 in the file shInterface.h.
u10 is a dummy argument.
u20 is a dummy argument.
v10 is a dummy argument.
v20 is a dummy argument.
r is the place to store the coordinates in the modeling space
  of the point of the surface closer to the given point.
z is the place to store the coordinates in the modeling space
  of the vector holding as beginning and ending points the
  given point xp and the found point r.
dist on output will contain the distance between the given
     point and the point of the surface closest to it.
imes is a dummy argument.
iter is a dummy argument.
xinit is a dummy argument.
utl is a dummy argument.
distl is a dummy argument.
ftl is a dummy argument.
zerom is a dummy argument.
      The function returns SH_OK if successful and SH_ERROR
      otherwise.
```

```
/******************************************
loas1r

Purpose : Load a curve from a list of edges.
Synopsis :
  int                    /* R  : non-zero on error */
  loas1r(
```

```
         int   *n,       /* O  : number of nodes of the edge */
         int   *isn,     /* O  : number of arcs composing the edges */
         float *xln1d,   /* O  : pointer to the curve geom */
         int   *is       /* I  : id of the edge to be loaded */
  )
Description : This function sets the current curve of the program
              to the one corresponding to a given edge. Any edge
              corresponds to an integer id, and this function
              writes the edge geom on a given memory area.
        n on output will contain the number of nodes of
          the edge, that it will be always 2.
          A node is a bound point between two different
          parameterization intervals of the edge.
        isn is the number of arcs composing the edge, that is
            always 1.
        xln1d is the place to store the edge geom.
              On output, it will contain -1 and -1 in its first
              and secund location, meaning that the geometry
              recorded is a SHAPES geom. In the third location
              from xln1d it is written this SHAPES geom.
        is is the id of the edge to be loaded.
               The function returns SH_OK if successful and SH_ERROR
               otherwise.


/******************************************
loas2r

Purpose : Load a surfaces from a list of faces.
Synopsis :
  int                   /* R  : non-zero on error */
  loas2r(
         int   *keypa,   /* O  : number of patches composing the face */
         float *aps,     /* O  : pointer to the face geom */
         int   *isur,    /* I  : id of the face to be loaded */
         int   *n,       /* O  : number of nodes in the u direction */
         int   *m        /* O  : number of nodes in the v direction */
  )
Description : This function sets the current surface of the program
              to the one corresponding to a given face. Any face
              corresponds to an integer id, and this function
              writes the face geom on a given memory area.
        keypa is the number of patches composing the face, that is
              always 1.
        aps is the place to store the face geom.
```

```
            On output, it will contain -1 and -1 in its first
            and secund location, meaning that the geometry
            recorded is a SHAPES geom. In the third location
            from xln1d it is written this SHAPES geom.
        isur is the id of the face to be loaded.
        n on output will contain the number of nodes of
          the face in the u direction, that it will be always 2.
        m on output will contain the number of nodes of
          the face in the v direction, that it will be always 2.
               A node is a bound point between two different
               parameterization intervals.
               The function returns SH_OK if successful and SH_ERROR
               otherwise.


/******************************************
compu

Purpose : Find the closest vertex from a point to a curve.
Synopsis :
  int
  compu(
        int   *n,           /* I  : dummy argument */
        float  x[3],        /* I  : input point */
        int   *iss,         /* I  : dummy argument */
        int   *isn,         /* I  : dummy argument */
        float *xln1d,       /* I  : address of the geometry to be queried */
        float *ug,          /* O  : global parameter of the closest point */
        int   *imel,        /* I  : dummy argument */
        float *dist         /* O  : distance between the given point and the
                                    closer one */
  )
Description : This routine finds the minimum of the function
              DISTPS = || r - x || where x is a point on the 3d
              space and r a point on a given curve.
              The search is limited to a prescribed interval.
        n is a dummy argument.
        x is the address of the array containing the coordinates
          in the modeling space of the given point.
        iss is a dummy argument.
        isn is a dummy argument.
        xln1d is the address of the geometry to be queried.
              It has to be a 1D xCELL and not a xGROUP.
        ug is the value of the parameter corresponding to the
           point of the curve closer to the given point x.
```

```
              The value ug belongs to the range defined
              by the constants T0 and T1 in the file shInterface.h.
        imel is a dummy argument.
        dist is the distance between the given point and the
             found one.
              The function returns SH_OK if successful and SH_ERROR
              otherwise.


/******************************************
gmeshinit

Purpose : Begin the SHAPES session and load a geometric model from
          a file.
Synopsis :
  void                   /* R  : - */
  gmeshinit(
           char *file   /* I  : SHAPES file holding the geometric model */
  )
Description : This function begins the interaction with SHAPES and
              read a geometric model from a file.
              It hides to the client all the SHAPES details,
              and the client will simply access the geometric
              objects by their indices read from a topology file.
        file is the SHAPES file holding the geometric model.


/******************************************
aisclose

Purpose : End the SHAPES session.
Synopsis :
  void                   /* R  : - */
  aisclose(
           void
  )
Description : This function ends the interaction with SHAPES.


/******************************************
loadFromFile

Purpose : Load a geometry from a SHAPES file.
Synopsis :
  xINT                                /* R  : non-zero on error */
```

```
  loadFromFile(
               char       *file,      /* I  : file holding the
                                             geometry to be loaded */
               shGEOMETRY *geometry   /* I/O: geometry to store the
                                             loaded geoemtry */
  )
Description : This function loads a geometry from a SHAPES file and
              records it in a shGEOMETRY structure.
        file is the file holding the geometry to be loaded.
             It must contain the string ".gms" if and only if it
             holds a list of xGEOMs.
        geometry is the structure to contain the geometry and the
                 list of its topological elements.
                 On output, its fields are initialized as:
                 dataFile points to file;
                 geom (or geoms) points to the geometric model and
                      the other to NULL;
                 isGeom is T or NIL according to the model being
                      a single xGEOM or a list.
              The function returns SH_OK if successful and SH_ERROR
              otherwise.


/*******************************************
getTopology

Purpose : Process a geometric model and initialize the list of all
          its topological entities.
Synopsis :
  xINT                               /* R  : non-zero on error */
  getTopology(
              shGEOMETRY *geometry   /* I/O: geometry to be processed */
  )
Description : This function queries a xGEOM for the lists of its
              vertices, edges, faces.
        geometry is the structure containing the xGEOM to be
                 queried.
                 On output, its fields are initialized as:
                 geomVertices describes the list of the vertices
                      of the model;
                 geomEdges describes the list of the edges
                      of the model;
                 geomFaces describes the list of the faces
                      of the model;
              The function also initialize the global variables
```

```
              pointing to the list of vertices, edges and faces
              of the model.
              The function returns SH_OK if successful and SH_ERROR
              otherwise.


/******************************************
checkMidPoint

Purpose : It checks whether the point corresponding to the
          middle value of the parameter range of a curve
          is contained in another curve.
Synopsis :
  xINT                        /* R  : T if edge1 and edge2
                                      share the edge1 "middle
                                      parameter point */
  checkMidPoint(
                xGEOM edge1,  /* I  : curve whose "middle
                                      parameter" point is
                                      computed */
                xGEOM edge2   /* I  : curve searched for
                                      the point containment */
  )
Description : This function searches for coincidence of two
              curves sharing the same vertices.
        edge1 is the curve whose "middle parameter" point
              is to be calculated.
              It has to be a 1D xCELL.
        edge2 is the curve queried for containment of the
              "middle parameter" point of edge1.
              It has to be a 1D xCELL.
              The function returns T if the "middle
              parameter" point of edge1 is contained in
              edge2 and NIL otherwise.


/******************************************
createEdgeKey

Purpose : It associates to an edge an hashing key based on
          its vertices indices.
Synopsis :
  xINT                          /* R  : hashing key for edge */
  createEdgeKey(
                xGEOM edge,     /* I  : edge whose hashing key
```

```
                                        is to be computed */
                xINT hashShift  /* I  : multiply constant used
                                        in the hashing key
                                        computing */
  )
Description : This function assigns the same hashing code
              to edges sharing the same vertices.
        edge is the curve whose hashing key is to be
             computed. It is computed as
             min(v1,v2)*hashShift + max(v1,v2)
             being v1 and v2 the indices of edge vertices.
             It has to be a 1D xCELL.
        hashShift is the multiply constant used in the
                  hashing key computing.
             The function returns the hashing key of
             edge.


/******************************************
markEdges

Purpose : It marks the edges of a model in such a way
          to make recognizable who are coincident.
Synopsis :
  xINT                          /* R   : non-zero on errors */
  markEdges(
            shGEOMETRY *geometry  /* I/O : model to be
                                           queried     */
  )
Description : It marks the first processed of a set of overlapping
              edges as master and all the other ones as slave.
              It attachs to each master edge an incremental
              index attribute, and to each slave the same index
              as its master.
        geometry is the model to be queried.
                 On output its 1skeleton are marked as master
                 or slave and have attached a suitable index
                 attribute.
              The function returns NIL if successful and T
              otherwise.


/******************************************
markVertices
```

```
Purpose : It marks the vertices of a model in such a way
          to make recognizable who are coincident.
Synopsis :
  xINT                              /* R   : non-zero on errors */
  markVertices(
               shGEOMETRY *geometry  /* I/O : model to be
                                            queried     */
  )
Description : It marks the first processed of a set of overlapping
              vertices as master and all the other ones as slave.
              It attachs to each master vertex an incremental
              index attribute, and to each slave the same index
              as its master.
        geometry is the model to be queried.
                 On output its 0skeleton are marked as master
                 or slave and have attached a suitable index
                 attribute.
              The function returns NIL if successful and T
              otherwise.


/*****************************************
writeTopology

Purpose : Write the topology of a geometrical model in a format
          that Mesh++ can read.
Synopsis :
  static xINT                         /* R  : non-zero on error */
  writeTopology(
                shGEOMETRY *geometry,  /* I  : geometry to be queried
                                              for its topology */
                char       *file       /* I  : file to write to */
  )
Description : This function processes the lists of the topological
              elements of a geometry and outputs them in the format
              requested by Mesh++.
        geometry is the structure containing the geometrical model.
        file is the path of the file to write to. If file already
             exists, it is overwritten.
                 The function returns SH_OK if successful and SH_ERROR
                 otherwise.


/*****************************************
shInit
```

```
Purpose : Initialize a geometry structure to describe the
          geometric model recorded in a SHAPES file.
Synopsis :
  xINT                          /* R  : non-zero on error */
  shInit(
         xCHAR      *file,      /* I  : SHAPES file containing the geometry */
         shGEOMETRY *geometry   /* I/O: structure to hold the geometry */
  )
Description : This function loads a geometric model from a file
              and initialize a geometry structure to describe it.
              This is accomplished by calling the loadFromFile
              ang getTopology facilities.
        geometry is the structure to describe the geometrical model.
                 On output, the structure will contain:
                 the model read from file in the field geom or geoms;
                 the list of its vertices in the field geomVertices;
                 the list of its edges in the field geomEdges;
                 the list of its faces in the field geomFaces;
                 a pointer to the string "file" in the field dataFile.
                 The field isGeom is 1 if file contains a single
                 xGEOM and 0 otherwise. In the former case, the model
                 address is recorded in the field geoms, while in the
                 latter one it is recorded in the field geom.
        file is the pathname of the file to be used.
              The function also initialize the heaps memory to
              store the SHAPES objects.
              The function returns SH_OK if successful and SH_ERROR
              otherwise.


/******************************************
shClose

Purpose : Free the memory used to allocate a geometrical model and
          terminates the interaction with SHAPES.
Synopsis :
  xINT                            /* R  : non-zero on error */
  shClose(
          shGEOMETRY *geometry    /* I  : geometry holding the current model */
  )
Description : This function deletes all the objects used to describe
              a geometrical model and terminates the SHAPES session.
        geometry is the structure to be freed;
              The function returns SH_OK if successful and SH_ERROR
```

otherwise.

```
/****************************************
exportGeomToOffFile

Purpose : Dump out a geom to a Noff file.
Synopsis :
  void                              /* R  : - */
  exportGeomToOffFile(
                      xGEOM geom,   /* I  : geom to be dumped out */
                      char *file    /* I  : file to be written */
  )
Description : This function writes a geom in Noff format.
        geom is the geometry to be dumped out.
             Are allowed only 2dimensional xGEOMs.
        file is the pathname of the file Noff to be written.
             Its extension must be .off.


/****************************************
length

Purpose : Compute the length of a vector of a 3D space.
Synopsis :
  float                  /* R  : the length of the vector */
  length(
         float *vector   /* I  : vector to be queried for its length */
  )
Description : This function calculates the lenght of a vector.
        vector is the address of the array containing the vector.
              The function returns the length of the vector.


/****************************************
euclideanDistance

Purpose : Compute the euclidean distance between two points of a
          3D space.
Synopsis :
  float                          /* R  : the euclidean distance between p1 and p2 */
  euclideanDistance(
                    float *p1,   /* I  : first point */
                    float *p2    /* I  : secund point */
  )
```

```
Description : This function calculates the distance between two
              points.
        p1 is the address of the array holding the coordinates of
           the first point.
        p2 is the address of the array holding the coordinates of
           the secund point.
              The function returns the distance between p1 and p2.


/******************************************
addPointsToGeom

Purpose : Join new vertices to a geom.
Synopsis :
  xGEOM                        /* R  : modified geometry */
  addPointsToGeom(
                xGEOM  g,      /* I  : geom to be modified by adding vertices */
                xREAL *points, /* I  : points to be joined to the geom g */
                xINT   n       /* I  : number of points to be joined to g */
  )
Description : This function performs a geometric union between a
              geom and a given set of points.
        g is the geometry to be modified.
        points is the address of the array holding the coordinates
               in the modeling space of the points to be joined to
               the geom.
        n is the number of points to be joined to the geom.
               The points array has 3*n elements.
               The function returns a geometry (a xGROUP)
               representing the union between the original geom g
               and the given points.


/******************************************
createAttr

Purpose : Create a new attribute of an already existent attribute
          specifier.
Synopsis :
  xATTR                   /* R   : created attribute */
  createAttr(
           xCHAR *specId, /* I   : specId of attribute
                                   specifier for the
                                   new attribute */
           xADDR  value   /* I   : address to be
```

```
                                        stored in the new
                                        attribute */
  )
Description : This function creates a new attribute storing
              a given address value.
        specId is the identifier string for the attribute
               specifier of the attribute to be created.
        value is the address to be stored in the data
              structure representing the attribute.
               The function returns the newly created
               attribute.


/*****************************************
attachIndexAttr

Purpose : Attach to a geometry an index attribute having
          value id.
Synopsis :
  xINT                       /* R   : non-zero on errors */
  attachIndexAttr(
                xGEOM g,     /* I/O : geom to be marked
                                        with the index
                                        attribute */
                xINT  id     /* I   : integer value of the
                                        index attribute to
                                        be attached */
  )
Description : This function creates an attribute of type
              index holding a given integer value, and
              associates it to a given geometry.
        g is the xGEOM to be marked with the index
          attribute
        id is the integer value of the index attribute
           to be associated to g.
          The function returns NIL if successful,
          and T otherwhise

/*****************************************
getIndexVal

Purpose : Get the integer value of the index attribute
          of a given geom.
Synopsis :
  xINT                  /* R   : index value of g */
```

```
  getIndexVal(
              xGEOM g    /* I   : geom to be queried */
  )
Description : This function queries a xGEOM for its
              index attribute, and this attribute
              for its integer value.

        g is the xGEOM to be queried.
          The function returns the index value of the
          geometry.


/******************************************
isMaster

Purpose : Query a geometry for its master attribute
Synopsis :
  xINT              /* R   : T if g is a master */
  isMaster(
           xGEOM g  /* I   : geom to be queried */
  )
Description : This function finds whether a geom is
              a master or a slave.
        g is the xGEOM to be queried.
          The function returns the T if the masterType
          attribute attached to the geom is master,
          and NIL otherwise.


/******************************************
mark

Purpose : Attach to a geometry an existing attribute.
Synopsis :
  xGEOM                 /* R   : the modified geometry */
  mark(
       xGEOM  geom,     /* I/O : the geometry to be
                                 marked with the
                                 attribute */
       xCHAR *string    /* I   : the string identifying
                                 the attribute */
  )
Description : This function associates to a geom an
              existing attribute identified by a
              string.
```

```
geom is the xGEOM to be marked with the
     attribute.
string is the identifier for the attribute
       to be attached.
       Currently supported attribute are:
       "master" -> master attribute
       "slave"  -> slave  attribute
      The function returns the modified geom
```

References

[1] Luca Formaggia, Giovanni Delussu - Mesh++ input description [VIVA-1998-2-10]