A Multiblock Mesh Manager

Olof Runborg,
Kungl Tekniska Högskolan,
100 44 Stockholm, Sweden

Abstract

In this report we address the development of a manager for structured multiblock grids. We use a method based on object-oriented data structures. The information about the topology and geometry are separated to minimize the interdependencies between blocks and the effects of operations on the geometry. Such a framework allows for widely different ways of describing the geometric information. The data structure also gives complete and readily accessible information about the topology of the multiblock grid.

1 Introduction

The aim of this subtask has been to develop a manager for structured multiblock grids. The focus of the development has been on the block topology, of which the manager should have full control. The manager should also be easy to expand, in particular with new ways to describe the geometry of blocks in the grid.

Typically grid generation involves a good deal of trial and error. Consequently the mesh manager should be flexible, and allow the user to freely edit, add and delete blocks and grids on different levels. One can distinguish several different aspects of the multiblock grid that the user should be able to modify.

First, there must be operations to edit a block's *geometry*, i.e. its (continuous) distribution in space. This could mean for instance changing the shape of an edge or a surface. Since blocks stick together, changing the geometry of one block, usually means that some of its adjacent blocks' geometry must change too.

Moreover, it should be possible to change the *topology* of the multiblock grid. By "topology" we mean how the different blocks are connected to each other. The topology is a property of the whole grid, not of the individual blocks. It changes when for instance a block is split into two, when new blocks are added or old ones deleted, etc.

The *mesh* should also be controlled by the user through appropriate operations. The mesh is considered to be the evaluation of the geometry at a discrete set of points. Some typical mesh characteristics that could be modified, are the mesh size (number of points) and the mesh density (stretching parameters). Because of the common constraint that grid lines should be continuous across block boundaries, changes in one block generally propagate to other blocks as well.

In addition, the user should be permitted to change the *geometry representation* of a block. The geometry representation is the set of parameters and the mathematical expression which describe the geometry. Sometimes edges are best described by B-splines, sometimes with straight lines or arcs. The user should have the choice. In particular this is important for the physical boundaries where a precise geometry description is often necessary.

Finally, it is very important that the grid generator allows one block to meet with multiple other blocks along one boundary. This may seem obvious, but in principle it is not necessary; one can always manage without this feature, by splitting up the larger block into many smaller ones. In practice, however, the number of blocks needed for a realistic problem becomes very high.

It shall be noticed that few of the operations described above affect only one single block; usually their effect spreads to numerous other blocks. This is the heart of the block management problem and the reason why a block topology manager is needed; it is not enough to handle only the blocks' geometry; the topology must be managed too. Keeping track of data in a way that gives the user full flexibility, is a formidable book-keeping problem. Finding a good data structure is vital.

In this report we will describe an object-oriented technique for multiblock grid management. It is a development of earlier work at KTH described in [Oppelstrup93] and [Runborg92]. We start by showing how the grid topology can be described with a directed acyclic graph (DAG). Next, we choose what classes to use, which comes naturally given the DAG description. We proceed by giving some details on the contents of the classes and how new classes can be added. Also, we say a few words about how compound objects can solve the problem with blocks of different sizes meeting. After a short introduction to the actual implementation, which is built on the TCL/TK package, we show results exemplifying how the data structures can be utilized. We sum up our experiences in a concluding section.

2 Method

We use a method based on an object-oriented data structure. There are two main ideas behind the approach. First, information about topology and geometry should be separated. In this way, operations on the topology can be isolated and written in a general, straightforward manner. Second, information about the geometry of the different parts of the multiblock mesh should be kept apart. This minimizes interdependencies between blocks and the effect of operations on the geometry becomes more predictable. It also makes it easy to use different geometry representations in different places, and to add new types of representations.

Our implementation was made in C++ and below we will frequently use C++ terminology when describing it. See [Stroustrup91] for a complete

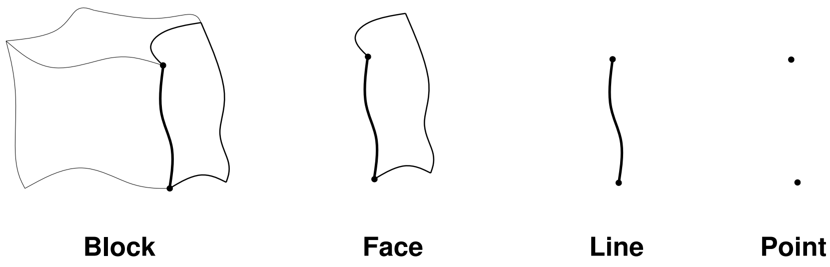

Figure 1: The primitive objects.

reference to the language.

2.1 A graph representation of the grid topology

The topology of the multiblock grid can be represented by a DAG. This will be shown here.

We start by considering four *primitive object types*, namely *block*, *face*, *line* and *point* (see Figure 1). On those we impose the following topological restrictions:

- a line has two end points,
- a face has four edges (where an edge is a line object),
- a block has six faces.

Next, we observe that there is a hierarchical relationship between the primitive objects. Objects that are part of another object (such as a line that is the edge of a face) can be seen as the *children* of the other object. Conversely, an object that contains other objects (such as a block that has six faces) can be seen as the *parent* of the other objects.

It should be noticed that the children of an object constitutes its boundary. Indeed, the object itself can be considered to represent only the inner points of its geometry. This thought will dictate the way the geometry is stored (see below).

Letting the nodes in the DAG represent the primitive objects, and the arcs go from parents to children, we get a representation of the topology. Examples are found in Figure 2.

There is one more piece of information that is needed for a complete description of the topology: the orientation of the objects. Since the orientation of an object can be different when seen from different parents, the information will be tied to the arcs of the DAG, rather than to the nodes. If we regard the orientation parameter as a weight, we get a weighted DAG.

2.2 Classes

Using the topology graph representation as a starting point, our next step is to create a set of C++ classes. These classes will contain all information

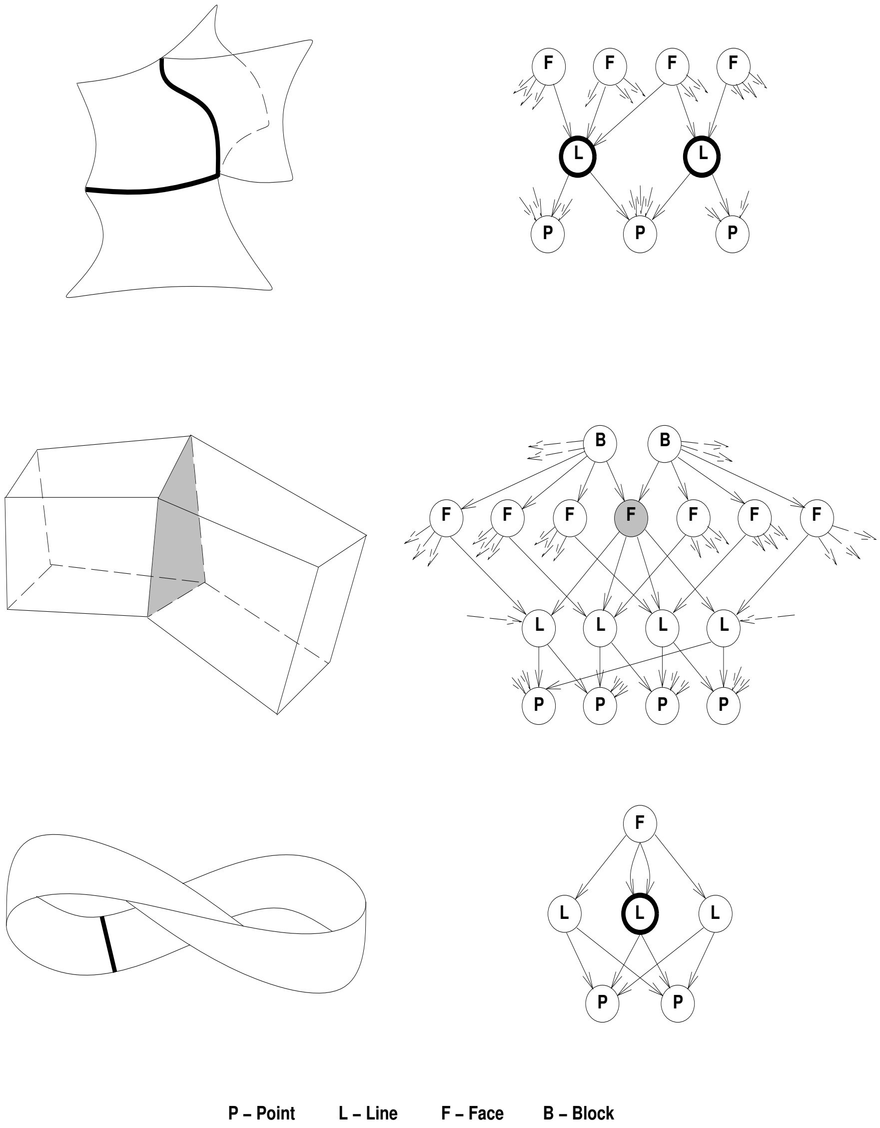

Figure 2: Some examples of topology graphs. Note that for each type of object the number of children is fixed, but, as in the case of the Moebius strip at the bottom, they do not have to be distinct.

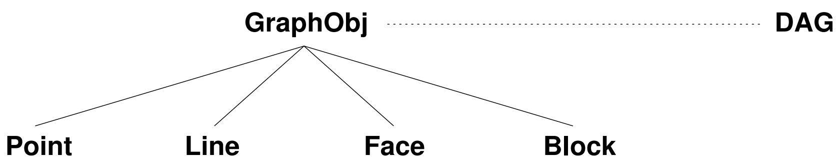

Figure 3: The initial class inheritance hierarchy.

needed about the multiblock grid, and the member functions through which the grid can be accessed and manipulated.

One C++ class was created for each of the primitive object types, i.e. a Point, a Line, a Face and a Block class. These primitive classes were all derived from the same abstract base class, called GraphObj. In addition we created a DAG class, which was made a friend of the GraphObj class. Thereby the following division of responsibilities between the classes could be made:

- the primitive classes Point, Line, Face and Block contain the geometry and mesh information about the object, and handles all related operations,
- the GraphObj class contains the topology information about the object,
- the DAG class handles all operations related to the topology.

The fact that topology information is stored in the GraphObj, but managed through the DAG, is actually a technicality. For all practical purposes, the topology can be thought of as being handled by the DAG. Hence, we have achieved a separation of geometry and topology information. The class inheritance hierarchy at this stage is presented in Figure 3. Later this will be expanded with variations of the primitive classes.

2.3 Contents of the classes

The contents of the classes are of two types: data and member functions. The data itself will in general be encapsulated within the classes, isolated from the rest of the program. Instead the member functions will work as an interface to the data. We will now go through the grid generation objects' data members and member functions, grouping them by the sort of information they carry.

Topology The topology information is the same for all classes, regardless of primitive type. Therefore it can be factored out and stored in the GraphObj class, which is the common base class for all objects. The following items must be stored:

- list of parents,
- list of children,
- orientation of each child.

For the Point class, the children list will be empty and for the Block class the parent list will be empty.

Each primitive class has a well defined reference configuration, detailing how the children's coordinate systems should be oriented relative to each other. The orientation parameter specifies how the actual orientation of a child's index coordinate system differs from the reference. A line can have two different orientations (forward or backward), while a face can have eight different orientations corresponding to the group of symmetries of the square.

The interface to the topology is handled via the DAG class. There are a multitude of different functions available for finding object dependencies, reading the parent/children lists, checking whether a set of objects can form a face or block, etc. This is an area where new functions are to be implemented, when need arises.

Geometry The class must contain a geometry representation, i.e. some data members which together with some algorithm/mathematical expression describe the object's geometry. The representation is arbitrary, since it will be hidden from the outside of the class.

The geometry representation should use its own, *local index coordinate system*. Moreover it should only be concerned with the *inner points* of the object's geometry, and not make any assumptions about the boundary; the geometry of the boundary is governed by its children. In this way we reach a separation of the geometry information pertaining to different parts of the multiblock grid; the description of the total geometry is split up into disjoint descriptions of the inner geometry of the nodes in the topology graph.

The primitive classes Point, Line, Face and Block have a minimal geometry representation. The Point uses three coordinate values (x, y, z). The Line stores no data members at all, but makes a direct linear interpolation between its end Points. Similarly, the Face and the Block stores no data members of their own, but they use standard transfinite interpolation [Eriksson85] between its edges (faces) to describe their inner geometry.

There is no member function working as an interface to the geometry. The interface to the mesh is used instead (see below). An interface to the geometry would seem necessary since the parent needs information about its boundary (i.e. its children's geometry) to be able to supply a continuous geometry description for arbitrary boundaries. For instance, except for the Point, the primitive classes use *only* the geometry of the children to describe their own geometry. However, the mesh interface is usually enough. A class could also disregard its children, but then its geometry would only be continuous at the boundary for a specific set of children.

Mesh The class should contain a meshing scheme, the generated mesh and a set of member functions as interface to the mesh.

The meshing scheme is arbitrary. It should use the geometry representation discussed above, and generate mesh points in the inner region of the object. The mesh points thus generated are stored inside the object as data members. By analogy with the disjoint geometry descriptions, we have that mesh points are only stored in one place, which allows us to avoid the consistency problems related with storing the same mesh point in multiple places.

The meshing is controlled via a meshing function taking the size of the mesh, and an orientation parameter, as arguments. To further control the meshing, it would be possible to also include an optional Stretcher object among the object's data members. The Stretcher object would contain some stretching function to alter the characteristics of the mesh. This has yet only been implemented for the Line type objects.

The interface to the generated mesh consists of a member function taking (integer) indices and an orientation parameter as arguments and returning the mesh point as a three dimensional vector. Each individual mesh point can also be set to an arbitrary vector with a similar function.

In a great many cases the mesh points' coordinates are enough information to complete the geometry of the parent. In particular, for linear and basic transfinite interpolation it is sufficient. This is the only interface that has been implemented at present. To get smoothness across block boundaries, some information about derivatives at the mesh points is necessary in general, but smoothness can also be obtained by a global smoothing operator which moves grid points.

2.4 Adding new classes

The primitive classes being established, the next step is to expand the class inheritance graph with classes using more advanced geometry representations. Instead of just the simple Line class (a straight line), we would like to add for instance a Bspline and an Arcline (a segment of a circle). Similarly for the Face we would like the alternatives Bspline surface, Bezier surface, and for the Block we might want to solve an elliptic differential equation to get the grid points inside the block, instead of using transfinite interpolation.

New classes are derived from the existing primitive classes. All necessary topology information has already been isolated in the GraphObj class. Therefore, the only new code that has to be produced when creating the new classes, is the code handling the geometry representation and the meshing scheme. This done, the new classes can be used freely together, anywhere in the grid, just as the original classes.

In the future new classes should be added as the demand for new types of lines, faces and blocks develops. Creating classes with complicated geometry representations should be simplified by the fact that there are many code packages available already for describing line and surface geometry. The packages could be used as basis and a C++ wrapper could be written around them. In Figure 4 an example of an expanded class inheritance graph is shown.

2.5 Super objects

So far we have implicitly assumed that blocks that meet have faces that match exactly. To remove this restriction, we have introduced a new

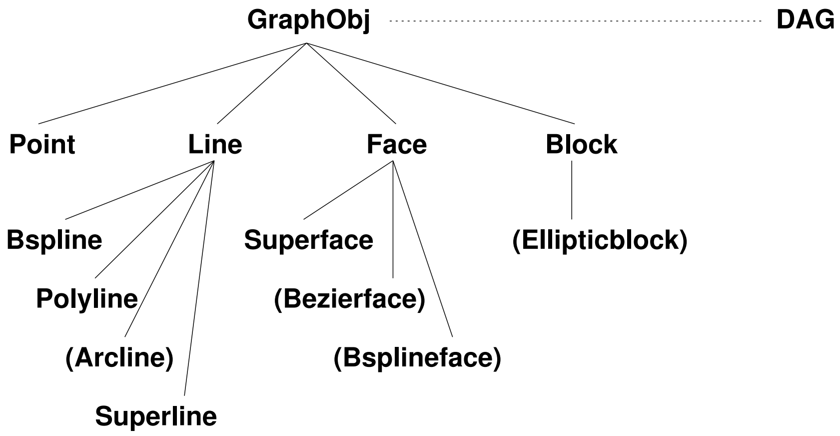

Figure 4: The expanded class inheritance hierarchy. Classes within parentheses have not yet been implemented.

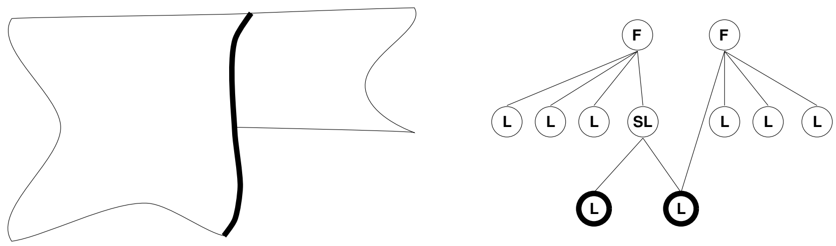

Figure 5: A configuration requiring a Superline.

variant of objects: compound objects or *super objects*, see Figure 5. We need two types: the Superline and the Superface. The Superline has other lines as children. The children could be any type of line (ex. Line, Bspline or even other Superlines) provided they form one long continuous line. The Superface has other faces as children. These children should be a patchwork of faces, together forming a face with exactly four edges.

The super objects are fundamentally different from other objects, in that their children do not constitute the boundary, but rather the inner points. Yet, a part of their functionality can easily be implemented, due to the strict encapsulation of geometry data. A case in point are the functions getting/setting mesh points, which are handled by just passing on the function arguments to the correct child.

A few functions have been disabled for super objects because they would be ambiguous. Meshing a super object is not allowed; its children must be meshed separately. A query about whether a super object is a physical boundary, will not return a well-defined answer.

3 Implementation

The interface to the user is built on a public domain package called TCL/TK. TCL (pronounced “tickle”) is a command line interpreter with a Unix shell-like look and feel. The functionality of a multiblock manager is obtained by adding new commands to the TCL command interpreter. The new commands are written in C++ and linked in with the TCL kernel. Basic commands for creating, deleting and meshing objects have been implemented this way, together with a number of different geometric variations of the primitive classes, as shown in Figure 4. TCL allows the user to write his own scripts and procedures for the mesh manager. Some of the functions written with pseudo-code in the next section (remesh, disconnect, join, addfaces) can actually be written as TCL procedures using almost the same wording. TK is a tool for easy creation of Xwindows-based graphical interfaces to TCL and hence to the mesh manager. For a full guide to the TCL/TK toolkit we refer to [Ousterhout94].

4 Applications

In this section we will present a few functions which were implemented using the object-oriented data structure. We will concentrate on a limited set of functions, which would have been difficult to realize, had the data structure not explicitly handled the topology. They are general functions that can handle any block topology and geometry.

4.1 Definitions

In the description of the functions below we will use the following notation for certain sets of objects. For a given object a we define:

all objects The set of all objects in the topology graph: G

parents The set of objects that are parents to object a: $P(a)$

children The set of objects that are children to object a: $C(a)$

upward dependency The set of objects above object a in the topology graph, ordered such that the child of an object in the set never comes before the object itself:

$$U(a) = \{b \in G \mid b \in P(a) \text{ or } \exists c \in P(a) : b \in U(c)\},$$

$$U(a) = \{b_1, \ldots, b_n\},\ b_j \in C(b_i) \quad \Rightarrow \ j > i.$$

downward dependency The set of objects below object a in the topology graph, ordered such that the parent of an object in the set never comes before the object itself:

$$D(a) = \{b \in G \mid b \in C(a) \text{ or } \exists c \in C(a) : b \in D(c)\},$$

$$D(a) = \{b_1, \ldots, b_n\},\ b_j \in P(b_i) \quad \Rightarrow \ j > i.$$

Note, all the sets are ordered, although the order is only significant for the upward/downward dependency sets. The notation A_i will signify element i of set A and $|A|$ the the total number of elements in A. Upward dependency for a set is defined as $U(A) = \cup_{a \in A} U(a)$, with the ordering constraint exactly as in the single object case (downward dependency analogously).

4.2 Remeshing

When the geometry of the multiblock mesh changes, it needs to be remeshed to get an updated mesh, evaluated at the new geometry. Typically, a geometry change is local and only a fraction of the multiblock mesh actually has to be recalculated. It is desirable to find this part, and to keep remeshing at a minimum.

Supposing that the geometry changes in one object, this object itself would obviously have to be remeshed. Moreover, since an object often constitutes the boundary of other objects, its parents, these objects' geometry will also be altered, and thus they also need to be remeshed. For instance, when the shape of a line is modified, not only does it have to be remeshed itself, but also the face that it is an edge of and the block that the face is part of. As the communication of the geometry from one object to another goes via the mesh interface, the remeshing also has to be done in the right order: the line first, then the face and lastly the block. In the general case, the objects that need to be remeshed is the upward dependency of the first object, and the constraints on the remeshing order is met if the upward dependency set is traversed in the reverse order. Hence, the algorithm for updating the multiblock mesh after a geometry change in object a would be:

procedure remesh_after_geometry_change (a) {
    remesh (a)
    for $i = |U(a)|$ **to** 1
        remesh $(U(a)_i)$
}

4.3 Changing topology

Changes in topology can be of different sorts. Adding and deleting objects are the most basic operations affecting the topology. Functions for splitting objects and for substituting objects are other examples. The difficulty in treating this kinds of functions is to ensure that the data structure, the topology graph, is not corrupted, e.g. left with loose ends or with nodes having the wrong number of children or wrong orientations.

When *creating* new objects, except in the case of a Point, the children of the new object must be specified. For a common Line there are no constraints on which children (end points) that are valid. For a Superline, however, all the lines given as children must be connected via points and for a Face they also have to form a ring. More complicated constraints must be observed for the children faces of Superfaces and Blocks. We have implemented functions to verify that an arbitrary set of objects actually is a valid set of children to use for creating a new object of a particular type. These functions also find what orientation parameters the children should have. Thus, the user can state the children in any order when creating new objects.

Deleting an object is not always possible. It is not legal if the object is part of another object, i.e. if it has one or more parents. This would leave the parents with one child too little. In different situations, it may be advantageous with different behaviours of the delete operation. For example, only delete the object specified or delete the object specified and as many as possible of the objects that are part of it (the downward dependency of the object).

Splitting an object involves both the topology and the geometry. Splitting is always made at grid points/grid lines/grid planes for Line, Face and Block type objects respectively. To split an object it is usually necessary to split its children first. For example, before splitting a Face horizontally, its two vertical edges must be split. The splitting is implemented hierarchically: a parent splits its children which split their children, etc. One split operation can create a great many new objects, each of which needs its own geometry representation. The geometry of the new objects must be based on the geometry of the original object. Consequently, each class must implement a splitting function that creates new geometry representations based on its own internal geometry description, often of the same type. For instance, a Bspline splits up into two new Bsplines and a new Point. A Face splits up into two new Faces, joined along a Polyline based on the mesh of the original Face, plus a few new objects created when the original Face's edges were split. Splitting has been implemented for all current classes. See Figure 6 and Figure 7 for examples of using splitting.

Functions for *substituting* one object for another can be very useful, for instance when objects are to be disconnected from each other or stuck together. The most interesting operation is to substitute an object in one or more of its parents. This demands caution, however, since there is a risk that the operation not only makes the parents' new set of children invalid, but also the grandparents' and grandgrandparents', etc. If we set the requirement that the new object and the old object should have exactly the same children, the parents are safe[1], but the older generations are still at risk. We use the middle picture of Figure 2 as an example. If we substitute one of the children of the shaded face for another line with exactly the same two end points as the original line, the shaded face will still be a valid face, but neither of the blocks will be valid any longer. For the blocks to remain valid, the line needs to be substituted in all three of its parent faces in this case.

To verify that a substitution operation of this type does not create invalid objects, we proceed as follows. Let S be the set of parents in which object a should be substituted for object b. (Usually, it is easier to specify an object c belonging to the upward dependency set of a and let $S = D(c) \cap P(a)$.) First, we check that a and b has the same children,

[1]This is usually an unnecessarily restrictive requirement, which is used here for the sake of simplicity.

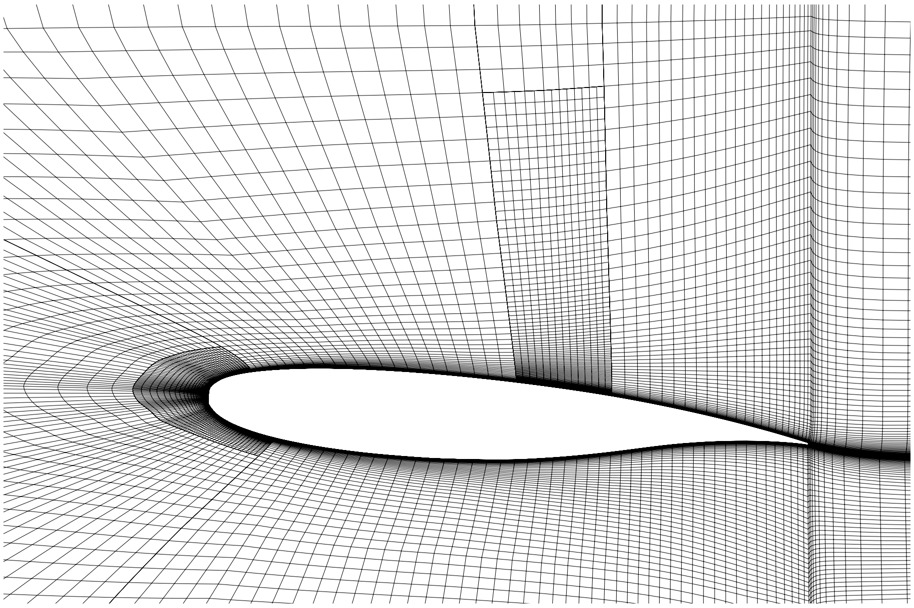

Figure 6: Example of splitting blocks around the DLR-F4 test case. The sub-blocks have been refined using the hanging nodes approach of DASA.

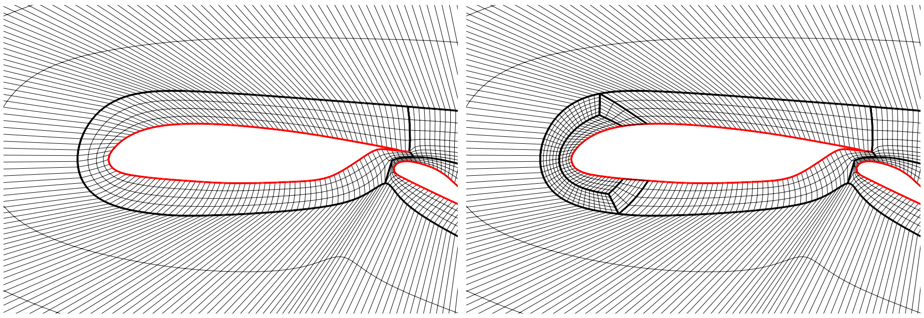

Figure 7: By using a series of split operations, more points can be added to a block without affecting the number of points in adjacent blocks. This introduces a number of singularities, however.

$C(a) = C(b)$. Next, we find the set of objects that will be affected by the substitution. This is the upward dependency set of S. Now, we argue that for a substitution to be valid, for each affected object, a must be substituted in *all* places in that object. Hence, the substitution is valid if and only if $(P(a) \setminus S) \cap D(U(S)) = \{0\}$. The substitution algorithm can therefore be written:

procedure substitute (a, b, c) {
    $S = D(c) \cap P(a)$
    if $(C(a) = C(b)$ **and** $(P(a) \setminus S) \cap D(U(S)) = \{0\})$ **then**
        foreach $x \in S$
            substitute_child_in_parent (x, a, b)
}

Together with a function for copying an object this substitute command can be used to *disconnect* an object from other objects. The new object returned by the copy command should have the same children as the original. The algorithm for disconnecting object p at child c would be (disregarding super objects for simplicity):

procedure disconnect (p, c) {
    substitute $(c, \mathbf{copy}(c), p)$
    for $i = |D(c)|$ **to** 1
        substitute $(D(c)_i, \mathbf{copy}(D(c)_i), p)$
}

A simple function for *joining* two objects at a specified parent object, can also easily be implemented using the substitute command. It requires that the two objects have similar downward dependencies, such as the objects returned by the disconnect routine, and it gives no control over how the objects should be oriented relative each other at the joint. The algorithm (disregarding superobjects for simplicity):

procedure join (p, a, b) {
    for $i = 1$ **to** $|D(a)|$
        substitute $(D(a)_i, D(b)_i, p)$
        delete $(D(a)_i)$
    substitute (a, b, p)
    delete (a)
}

Figure 8 demonstrates the use of the split, disconnect and join operations.

4.4 Mesh size compatibility

At the boundary between blocks, solvers usually require that grid lines should be continuous. This constraint implies that not all mesh sizes are valid; adjacent blocks must have the same number of grid points along the common boundary. If they have, we call the mesh sizes locally compatible. However, as long as there are unmeshed blocks in the grid, there may be,

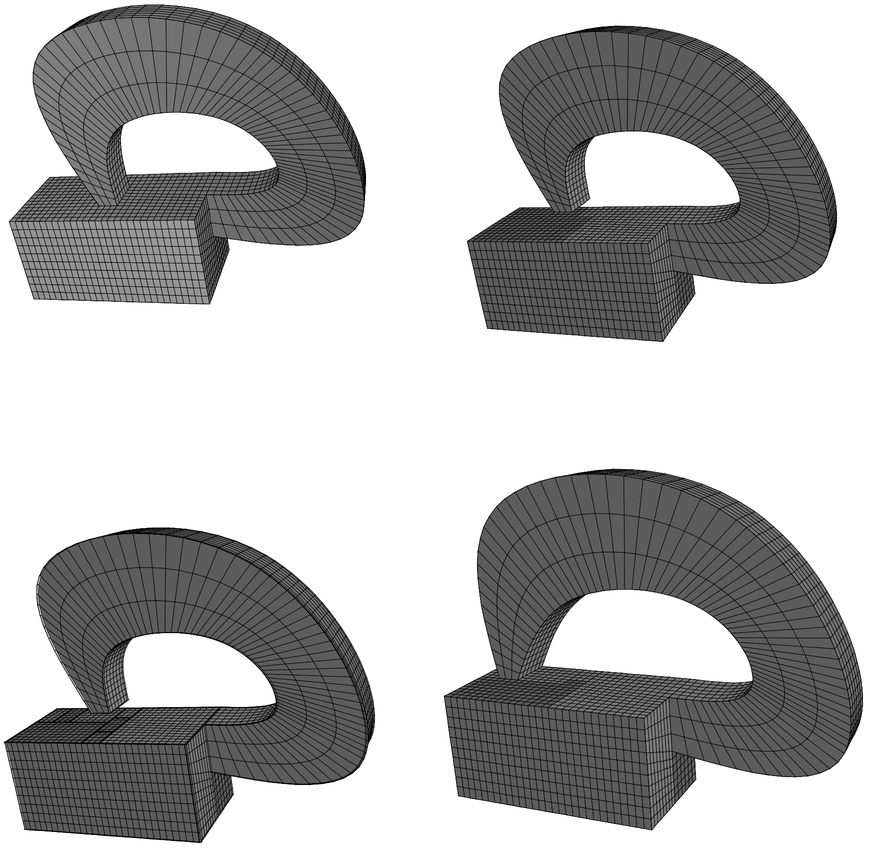

Figure 8: The curved block of the original grid (top left) is first disconnected (top right), then the upper face of the brick shaped block is split (bottom left) so that the curved block can again be joined (bottom right) with the other block, at a different place. Block connectivity information is updated with each operation.

among them, blocks for which there are no valid mesh sizes, even though all existing meshes are locally compatible with each other. If there is at least one combination of mesh sizes for the unmeshed blocks in the grid that leaves *all* mesh sizes locally compatible, we call the existing mesh sizes globally compatible.

Some solvers do accept that meeting blocks have different numbers of grid points. To allow for this kind of "hanging nodes" the blocks can simply be disconnected (see above) at the object where they meet. Then the blocks no longer impose any restrictions on each others' mesh sizes (see e.g. Figure 6).

To find the locally compatible mesh sizes of an object, the parents of the object itself and of all its children and grandchildren must be consulted. It is vital to sort out the objects that themselves are subject to change when meshing, and not let their current mesh size impose bogus restrictions on the new mesh size. The algorithm would read:

procedure meshsize_template (a) {
  $S = D(c) \cup a$
  foreach $x \in S$
    foreach $y \in P(x) \setminus S$
      Find what restriction y puts on the mesh size of x.
  Collect all mesh size restrictions on objects in S and
  summarize in a mesh size template for a.
}

Note that some care is necessary to handle super objects correctly.

The question of global compatibility of mesh sizes is dealt with by considering only the mesh sizes of the lines of the multiblock mesh. These mesh sizes immediately determine the mesh sizes of faces and blocks. A function has been implemented that, given a state of the multiblock mesh with certain objects meshed and others not, returns all line mesh sizes that are uniquely determined by the state, and all line meshes that are inconsistent with each other.

Let n be the number of lines in the multiblock mesh, x_i the number grid points of line i and $\boldsymbol{x} = (x_1, \ldots, x_n)^T$. The requirement of grid line continuity across block boundaries gives a set of conditions that has to be met. For instance, lines at opposite sides of a face must have the same number of grid points. Already meshed lines give further conditions. Let m be the total number of conditions. They will form an in general underdetermined linear system of equations,

$$A\boldsymbol{x} = \boldsymbol{b}, \qquad A \in R^{m \times n}, \qquad \boldsymbol{b} \in R^m.$$

Negative and non-integer solutions should be excluded.

Uniquely determined elements of $\boldsymbol{x}$ are those whose corresponding basis vector $\boldsymbol{e}_i$ is perpendicular to the null space of A,

$$\boldsymbol{e}_i \perp \mathsf{N}(A).$$

Let $P_{\mathsf{R}(A)}$ be the projection onto the range of A. Inconsistent conditions in A are those where

$$b_i \neq (P_{\mathsf{R}(A)}\boldsymbol{b})_i.$$

A simple and numerically stable way to get both the null space and the range of A in a convenient form, is through the singular value decomposition of A,

$$A = U\Sigma V^T.$$

Denote the column vectors of the orthogonal matrices U and V by $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$, respectively. Suppose r is the rank of A, i.e. the number of non-zero singular values. The number of degrees of freedom to choose mesh sizes will be $n - r$ and

$$\begin{aligned} \mathsf{R}(A) &= \operatorname{span}\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r\}, \\ \mathsf{N}(A) &= \operatorname{span}\{\boldsymbol{v}_{r+1}, \ldots, \boldsymbol{v}_n\} \end{aligned}$$

(see for instance [GolubvLoan89]). From this information the uniquely determined and the inconsistent mesh sizes can be established.

The possibility to check the global mesh size compatibility can be used to automatically mesh all objects that have not yet been meshed. This can be a laborious task to do manually, if the number of blocks is large and the topology is complicated. Since in general there is a positive number of degrees of freedom, the automatic meshing algorithm needs some hint as to what mesh sizes it should choose. We have used the following algorithm, that takes a value h representing the desired size of a mesh cell:

1. Sort all unmeshed lines according to length, shortest first. Unmark all of them and put them in a list.
2. Set current line to first unmarked line in the list.
3. Mesh current line with a number of grid points as close as possible to its length divided by h.
4. If this produces inconsistent conditions in the A matrix, unmesh line, let current line be next unmarked line in list and goto 3.
5. Mark all lines whose mesh size can now be uniquely determined.
6. If there are still unmarked lines in the list, goto 2.

The reason why the lines are first sorted according to length is that it has empirically proved to lessen the chance of the algorithm failing due to results generating negative or non-integer mesh sizes. Some results of the algorithm is shown in Figure 9.

4.5 Adding faces

When creating new multiblock meshes it can be a drawn-out process to add manually all block connectivity information. This can be automated to a certain extent. We have implemented a function for automatically adding faces given only a line skeleton. The function scans the multiblock topology and adds new faces where this is possible, i.e. where four lines form a ring. In general the function will find too many faces, but it is far less timeconsuming to filter out the wrong faces among a set of candidates, than creating all the right ones manually. The following algorithm is used, with a slight modification also to allow for super objects.

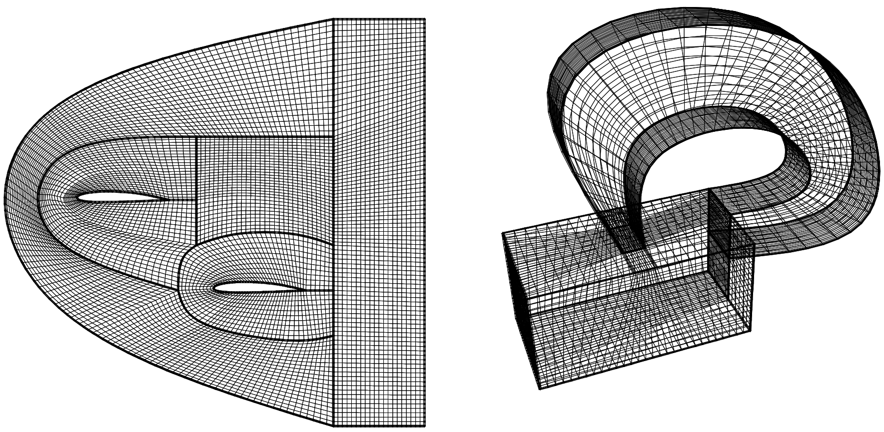

Figure 9: Result of the automeshing algorithm for two different configurations. The left mesh was generated without any constraints. For the right configuration the number of grid points was specfied at the face connecting the curved block with the brick shaped one; the remaining mesh was computed with the automeshing routine.

procedure search_for_faces $(first\_point, previous\_point, next\_line)$ {
  $FaceCandidate = FaceCandidate \cup next\_line$
  $next\_point = C(next\_line) \setminus previous\_point$
  if $(next\_point = first\_point$ **and** $|FaceCandidate| = 4)$ **then**
    create_new_face $(FaceCandidate)$
  if $(|FaceCandidate| < 4)$ **then**
    foreach $line \in (P(next\_point) \setminus FaceCandidate)$
      search_for_faces $(first\_point, next\_point, line)$
  $FaceCandidate = FaceCandidate \setminus next\_line$
}

procedure add_faces {
  foreach $a \in G$
    if (**primitive_type**(a) = **line**) **then**
      search_for_faces $(C(a)_1, C(a)_1, a)$
}

4.6 Smoothing

A smoothing modules has been linked to the program. It operates globally, and is able to smooth across block boundaries, thereby changing the geometry of the blocks. Mesh points belonging to objects with the boundary attribute set, will not be moved.

The smoother handles three-dimensional grids, i.e. block, face and line meshes. It is implemented directly in the mesh manager, taking advantage of the data structure. The algorithm it uses is very simple; the new location of a mesh point is the average of its closest neighbours' locations. However, the implementation should rather be seen as an example of how easy it is to add this type of functions to the manager. It is very general, handling singular points with any number of neighbours, isolated line meshes and non-anchored boundaries. In total, it required a programming time of only a few days.

5 Conclusions

The framework for a multiblock mesh manager has been built, using an object-oriented approach. The multiblock grid topology is represented by a graph, describing the natural hierarchical relationship between points, lines, faces and blocks. Each node in the graph corresponds to an object which contain the geometric information about that part and only that part of the grid. This data structure gives complete and readily accessible information about the topology of the multiblock grid. At the same time, it allows for widely different ways to describe the geometry of different parts of the grid. In the final section we have shown a few examples of how functions can take advantage of the data structure.

Furthermore, we have opened up a link to the data structure, by joining it with the TCL command line interpreter. Besides enabling the user to directly interact with the data, this gives him the possibility to write scripts and functions, using the full strength of TCL's control structures and variable handling.

Essentially, there are two disjoint areas for expansion: new classes with alternative geometry descriptions and new topology functions in the DAG class. We have focused on creating a skeleton program, containing some basic geometric classes, and an adequate set of general topology functions, on which the user can build more complicated functions, either in TCL or in C++, linking them directly into the program. It has been our ambition to make the mesh manager easily extensible, and in this effort we have been much helped by using an object-oriented language, such as C++.

References

[Eriksson85] *Eriksson, Lars-Erik, (1985) :* Practical three-dimensional mesh generation using transfinite interpolation, SIAM Journal of Scientific and Statistical Computing, **6** 712–741

[GolubvLoan89] *Golub, Gene and Van Loan, Charles F, (1989) :* Matrix Computations, The Johns Hopkins University Press, 2nd ed.

[Oppelstrup93] *Oppelstrup, Jesper et al, (1992) :* Development of 3D multi-block mesh generation tools, In Weatherhill, Nigel et al, eds., *Multiblock Grid Generation Results of the EC/BRITE/EURAM EUROMESH*, Vieweg, **44** 117–129

[Ousterhout94] *Ousterhout, John K., (1994) :* TCL and the TK Toolkit, Addison-Wesley, The TCL/TK distribution is available by anonymous ftp at `harbor.ecn.purdue.edu:/pub/tcl/sprite-mirror`.

[Runborg92] *Runborg, Olof, (1992) :* Object-oriented skeleton program for multiblock grid generation, TRITA-NA-E9255, NADA, KTH.

[Stroustrup91] *Stroustrup, Bjarne, (1992) :* The C++ Programming Language, Addison-Wesley, 2nd ed.