

MULTIPLE INTERSECTIONS OF FINITE-ELEMENT SURFACE MESHES

William M. Lira¹, Luiz C. G. Coelho², Luiz F. Martha³

¹Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil - william@tecgraf.puc-rio.br

²Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil - lula@tecgraf.puc-rio.br

³Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil - lfm@tecgraf.puc-rio.br

ABSTRACT

This paper extends a previously proposed algorithm for the intersection of finite-element surface meshes. The new version of the algorithm treats special cases of multiple intersections that were not dealt with in the original version. This paper also describes a class library, in the context of Object Oriented Programming, for the generic handling of several types of surface geometry. This library is incorporated to a geometric modeler, allowing its use for modeling complex models.

Keywords: finite-element surface meshes, multiple intersections, parametric surfaces

1. INTRODUCTION

Geometric modeling [1] and finite-element analysis [2] are important topics in the process of simulating Engineering problems, especially when the analytic solution is unknown or difficult to obtain (see, for example, Figure 1).

In geometric modeling, a way to build complex models consists in combining several simple surface patches constructed individually. This requires intersecting surface patches and trimming off exceeding patches resulting from surface intersection. When this modeling technique is adopted, the intersection problem between two surface patches becomes a very important issue and must be treated in an efficient and robust manner.

In realistic Engineering modeling, few cases of surface intersection can be analytically solved. The practical solution is to treat surface intersection by means of numerical techniques.

Numerical techniques belong to two basic types: *marching* and *subdivision* methods. The *marching* methods, also called *continuation* methods, compute intersection curves in three dimensions by marching in the direction of its tangent vector or higher derivatives to obtain points along the curve [3-5]. The *subdivision* (or *decomposition*) method computes trimming curves in two-dimensional parametric space by recursively refining the solution at each step [6].

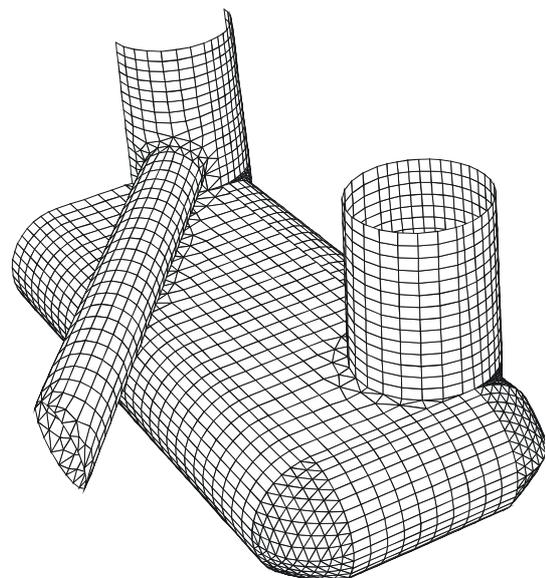


Figure 1. An example of complex geometric modeling.

In the context of finite-element modeling, not only surface intersection is important, but also mesh generation on the intercepted surfaces must be considered. One additional requirement is that finite-element meshes be compatible among the several surface patches. This means that a mesh generated on a surface patch has to conform to the mesh generated on adjacent patches.

Figure 2 shows an example of a compatible finite-element mesh that was generated using the methodology adopted in this work (note that only local modification were made to the original cylinders' meshes).

Several solutions for the surface-intersection problem work relatively well in many practical cases, but few consider the problem of compatible meshes. An exception is the work by Lo [7], which presents a simple algorithm for triangular-mesh intersection that automatically redefines the triangles, adapting them to the intersection curves detected. However, this solution works directly with the intercepting meshes and does not use any sort of surface parametric representation. This means that intersection points may not be located on the original surfaces, which is a serious drawback, especially in the case of multiple intersections.

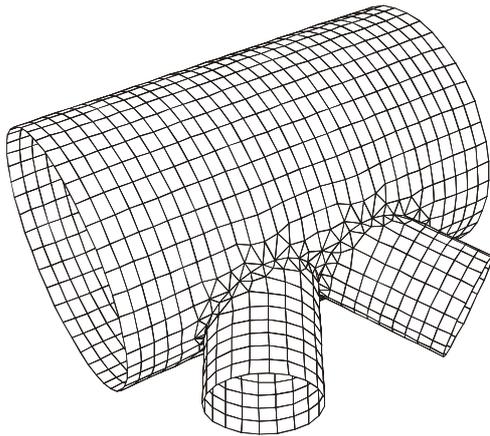


Figure 2. Compatible finite-element surface meshes.

The present paper extends a previously proposed algorithm [8] for the intersection of finite-element surface meshes that integrates the problem of parametric-surface intersection with mesh compatibility at intersection curves. The algorithm generates surface elements resulting from intersections with improved geometric quality, suitable for finite-element analysis. This new version of the algorithm treats special cases of multiple intersections that were not dealt with in the original version, due to geometrical and topological problems.

This algorithm is actually a scheme for parametric-surface intersection in which existing surface meshes are used as a support for the definition of accurate intersection curves. The geometric representations of the intersection curves

consist of B-splines defined by interpolation points that result from the intersection of the existing meshes and that are numerically relaxed to the surfaces. In this sense, intersection curves are defined in a discrete fashion, as opposed to representing them analytically. This discrete scheme avoids certain problems experienced by most modelers, such as inconsistencies between parametric representations of a point in the intersection of two or more surface patches.

The searches required for computing the intersection curves and for re-meshing the surfaces are supported by an auxiliary topological data structure whose main feature is that topological entities are stored in spatial-indexing trees, instead of linked lists. These spatial-indexing structures play a major role in the overall efficiency of the algorithm. The auxiliary data structure is defined in the parametric space of each intersecting surface and is also used to locally re-mesh the intersecting surface meshes.

However, the original version of this algorithm did not treat some special cases of surface intersection that may arise in multiple intersections. These cases, despite being seemingly simple, are very useful in practical finite-element surface modeling. For example, the original version of the algorithm did not handle situations in which intersection curves defined at a previous step of the modeling are intercepted by other surfaces.

Therefore, the purpose of this work is to present an extended version of this surface-mesh intersection algorithm, treating special cases that were not considered in its original version. In this new version, the methodology of the original algorithm and its central ideas have not been modified.

This work also presents a class organization, in the context of Object-Oriented Programming (OOP), of the data structure used in the surface-intersection algorithm. The main goal of this implementation is the generic handling of several types of surface geometry. In addition, this allows the use of the proposed algorithm by any geometric modeler. The requirements for the generic use of this library are that surfaces have two-dimensional parametric representations and that an initial finite-element mesh be defined on each surface.

This methodology could be implemented using currently available solid-modeling libraries, such as ACIS [9], Parasolids [10], Open CASCADE [11] and Pro/ENGINEER API Toolkit [12]. These libraries provide topological and geometric representations as well as Application Program Interface (API) functions, which are necessary for this type of modeling. However, they are expensive, include a large number of classes, and have long APIs.

Seeking to tackle these issues, the authors of the present paper have been involved for the last decade in the development of a modeling tool, called MG [8,13], which may also provide an appropriate environment for the implementation of the target methodology. One key aspect in this methodology is the integration of geometric modeling and automatic/adaptive finite-element mesh generation. This integration provides a consistent

conversion between the geometric model and the finite-element representation, and allows a fast prototyping of new concepts using relatively small pieces of software.

Section 2 summarizes the original algorithm for surface-mesh intersection. Section 3 describes the modifications in the algorithm for treating multiple intersections. In Section 4, the OOP class library is detailed. Section 5 provides application examples created by the MG modeler. Finally, Section 6 shows the conclusions of this work.

2. ORIGINAL ALGORITHM FOR SURFACE-MESH INTERSECTION

The surface-intersection algorithm proposed by Coelho [8] to perform the intersection between two parametric surface patches with meshes, A and B , computes the intersection curves and the new compatible meshes in three basic steps:

- I. Determination of the intersection points:
 - a. Compute and store the intersections of edges in A against faces in B ;
 - b. Compute and store the intersections of edges in B against faces in A .
- II. Determination of the trimming curves:
 - a. Link intersection points into polygonal lines representing the trimming curves;
 - b. Interpolate parametric curves through polygonal line points;
 - c. Compute new points with proper spacing on these curves;
 - d. Move these new points onto each surface.
- III. Topology reconstruction:
 - a. Determine the trimming regions by removing vertices and edges near the trimming curves;
 - b. Insert new edges over the trimming curves using the new points defined in Step II;
 - c. Triangulate the trimming regions on both surfaces;
 - d. Smoothen both meshes using original parametric descriptions.

A detailed description of the original version of the algorithm summarized in this section and of the methodology used by it can be seen in the original work by Coelho [8].

2.1 Determination of Intersection Points

To avoid testing all edges against all faces in Step I, the topological entities are stored in spatial-indexing trees. As edges and faces are curved in 3D space, it is necessary to use numerical procedures to determine the intersection points. At the end of Step I, edges in one mesh are paired with the faces they intersect on the other mesh, and vice-versa. For each edge/face pair, the parametric coordinates of intersection points are also stored. In the edges, this information is stored in a field called *intersection*, which is used to determine the trimming curves and to rebuild the topology. This field is the key to link both surfaces' data structure.

2.2 Determination of the Trimming Curves

In Step II, the trimming curves in parametric space are computed by linking and interpolating the intersection points obtained in Step I. The intersection curves are first obtained as polygonal lines, connecting the intersection points to produce polylines. To convert the polylines into continuous intersection curves, a piecewise cubic interpolation is performed.

After interpolation, the intersection curves are sampled to obtain uniformly spaced points, with spacing proportional to the size of the edges in the initial mesh. The sample points define the vertices on the intersection curves in the combined mesh. Although the interpolation points lie on both surfaces, the new sample points may not, and they must be translated back onto the surfaces by means of a procedure similar to the one used in Step I. The convergence is much faster in this case, due to the proximity of the starting positions. As a by-product, the parameter values for the sample points are also obtained.

2.3 Topology Reconstruction

In Step III-a, the trimming regions are identified. These regions are the faces of the topological data structure generated by the elimination of some edges. Step III-b consists of inserting edges that represent the trimming curves. In Step III-c, the trimming regions generated in Step III-b are triangulated using geometric approaches that guarantee mesh consistency.

To improve the shape quality of the faces generated by the intersection step, a smoothing technique is applied in Step III-d, where the parametric coordinates of each vertex are modified by an average between the adjacent coordinates. Boundary and trimming vertices are never moved during smoothing.

2.4 The Intersection Data Structure

A variant of the DCEL data structure [14], extended to link two surface finite-element meshes, is used to store surface patches. This data structure was also extended in the original version of the algorithm to handle topologies of curved elements, storing topological entities (vertices, edges and faces) in trees, instead of linked list or vectors. The use of trees allows for fast solutions to the searches necessary for the initial mesh assembly and for mesh reconstruction. The modified DCEL data structure is shown in Figure 3.

Vertices and edges are stored in B-trees [15], while faces are stored in R*-trees [16]. The vertices are inserted in a B-tree that searches points by their parametric coordinates.

The edges, which are considered straight in parametric space, connect pairs of vertices. Edges are oriented from the vertex with the smallest index to the vertex with the greatest index, and are stored in a B-tree that has these indices as search keys, as in the vertex B-tree.

The faces are inserted into an R*-tree that uses their 3D bounding boxes as keys [17].

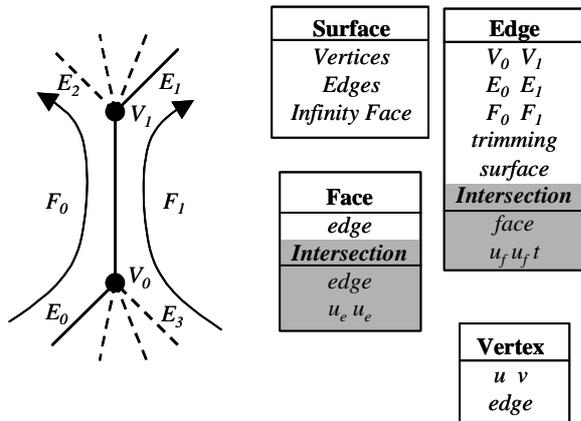


Figure 3. Modified DCEL data structure.

As previously mentioned, the edges and faces contain *intersection* fields used to determine the trimming curves. These fields are also the key to the topological reconstruction of two DCELs resulting from intersection. The geometric information of each intersection point is stored in pairs (u_f, v_f) and (u_e, v_e) in the *intersection* field.

3. MULTIPLE INTERSECTION

The original version of the current algorithm treats several cases of intersection between two surface patches. However, in the modeling of realistic Engineering problems, intersections of more than two patches must also be treated. The solution to this problem still uses the pairwise surface intersection approach of the original algorithm. New information is added to the modeling process to consistently treat previous intersections. Figure 4 illustrates the approach adopted for multiple intersections. The intersection of three surfaces (A, B, and C) is considered (Figure 4-a). In the first step (Figure 4-b), the modeler intercepts surfaces A and B. In the next step (Figure 4-c), surface C intercepts surface A, which in this case is modified by the previous intersection. Finally, surface C intercepts the modified surface B (Figure 4-d).

This section describes the accomplished modifications in the computational implementation of the algorithm, with the purpose of treating multiple intersections.

A special case dealt with in this work, and shown in Figure 5, refers to situations where an edge $e1$ of surface Sa intercepts exactly an edge $e2$ of surface Sb . As mentioned in the previous section, in Step I of the original algorithm the *intersection* field of edge $e1$ was filled only with one face of Sb (in this case, face $Sb1$ or $Sb2$), and vice-versa. In Step II, this field is used to link two topological data structures. This linking is used in the computation of intersection curves, traversing the intercepted faces in Sa and spreading the intersections by adjacent faces and intercepted edges, identified by the *intersection* fields. Since each *intersection* field stored only an edge or face at

a time, a situation could occur in which it was not possible to determine the intersection curves in Step II – that is, some edges or faces of the data structure could not be traversed by the algorithm. In this case, there was no guarantee that the intersection algorithm would reach the desired result.

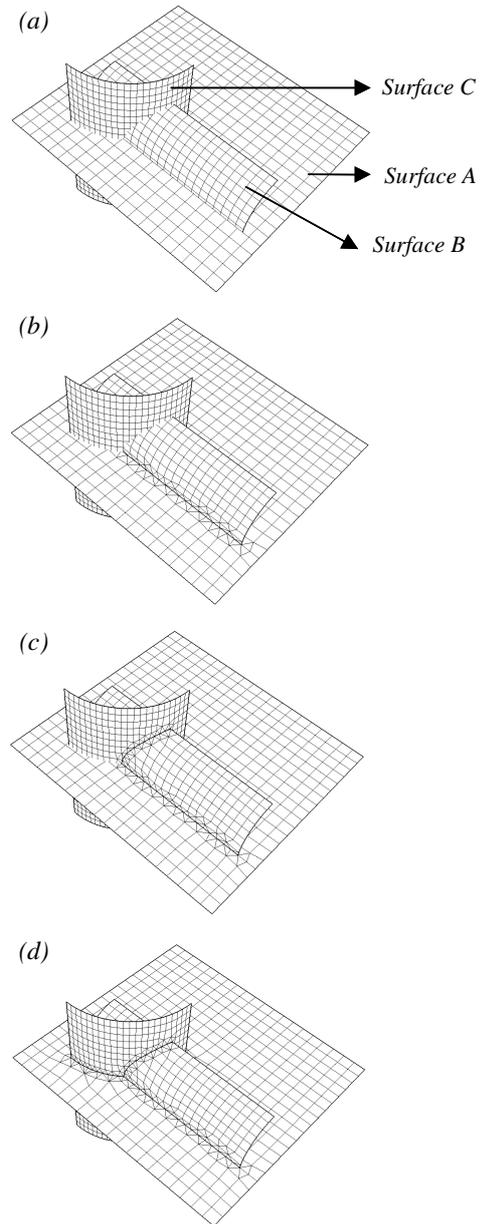


Figure 4. Pair-wise approach adopted for multiple intersections.

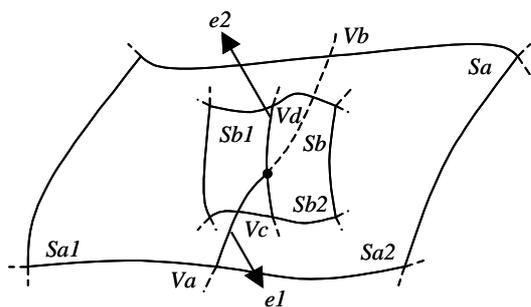


Figure 5. Special case: edge/edge intersection.

The solution to this problem is achieved by modifying the *intersection* field associated to edge entities of the topological data structure. This field no longer stores only a face intercepted by it. In this new implementation, this field stores a list with all faces intercepted by the edge, thus preventing some faces from not being traversed in Step II of the algorithm. Figure 6 illustrates a typical example in which an edge/edge intersection occurs and presents the resulting mesh consistently treated by the new version of the algorithm.

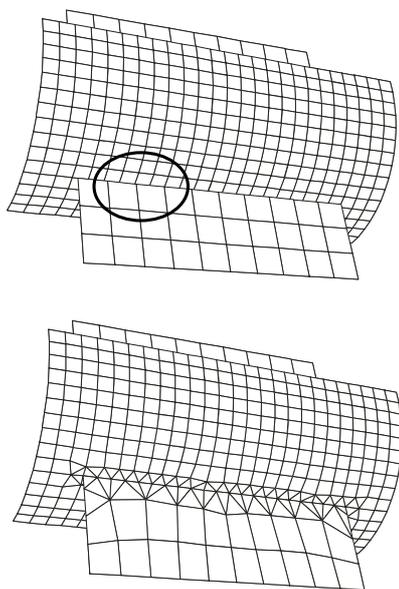


Figure 6. Example of mesh with edge/edge intersection.

A similar problem happens when an edge of a surface intercepts exactly a vertex on the other surface. The solution adopted for this problem is the same considered in the case of edge/edge intersection.

Another special case considered by the new version of the surface-intersection algorithm refers to situations in which the obtained intersection curve crosses other previously defined intersection curves. This case is complex because, in the generation of trimming regions, edges and vertices on previously defined intersection curves could be removed, which is not desirable.

The modification in the original algorithm to solve this problem consists in verifying, between Steps II and III, whether the intersection curve crosses previous intersection edges or vertices. When this occurs, the existing edges, rather than being removed, are just split. Figure 7 illustrates this special case.

In the modeling approach used in this work, the creation of complex geometric models requires the combination of several simple surface patches constructed one by one. In this case, the meshes obtained from the intersection between two surface patches could be inconsistent with a mesh on a third surface patch adjacent to the intercepted meshes, as can be seen in Figure 8.

To avoid this situation, the new version of the surface-intersection algorithm allows the automatic reconstruction of meshes on adjacent surface patches (see Figure 9). This is made using the same topological data structure presented above, with some adaptations. Steps I and II of the algorithm are not used, because it is not necessary to compute the intersection points and trimming curves. These steps are replaced by a procedure that inserts in the data structure the trimming curve points generated by the intersection of the two surfaces and that touch adjacent surfaces. In Figure 8, there is only one intersection point, P_i , that touches the adjacent surface, C . Two cases can occur: either the inserted point corresponds to a vertex already defined or the inserted point is on an edge. In the first case, the edges that frame into this vertex are removed, except for the constrained or boundary edges. In the second case, as in Figure 8, an edge is split.

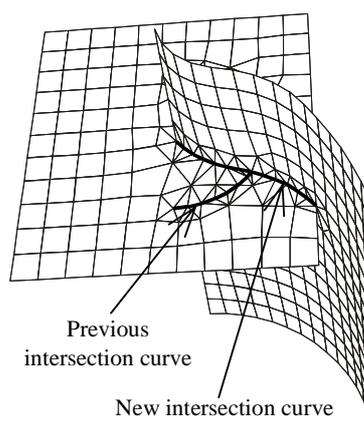


Figure 7. Example where a new intersection curve crosses a previously defined intersection curve.

Step III of the algorithm was adapted to perform the triangulation of the new regions obtained. Only Steps III-c and III-d are used because, in this case, there is no trimming curve and the region to be meshed is already identified.

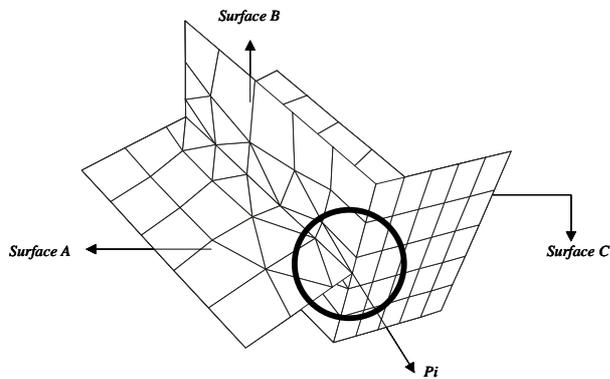


Figure 8. Rebuilding the mesh of an adjacent surface: the problem.

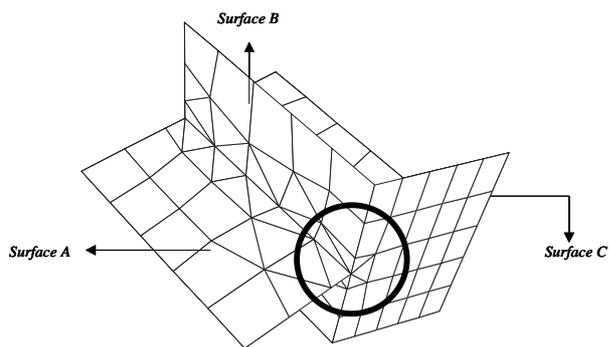


Figure 9. Rebuilding the mesh of an adjacent surface: the solution.

4. OOP CLASS LIBRARY

The surface-intersection algorithm described in previous sections was implemented as an OOP class library using the programming language C++. Treating the algorithm as a library allows its easy use by a great amount of geometric modelers. There are two requirements to incorporate this class library into a modeler: parametric representation of surfaces and the definition of finite-element meshes on surfaces.

The class structure adopted is relatively simple. Its classes are shown in Figure 10, which also illustrates the communication flux between the class objects. There are classes whose objects represent each of the topological entities of the data structure. The *dcelVertex*, *dcelEdge* and

dcelFace classes describe, respectively, vertices, edges and faces in the data structure.

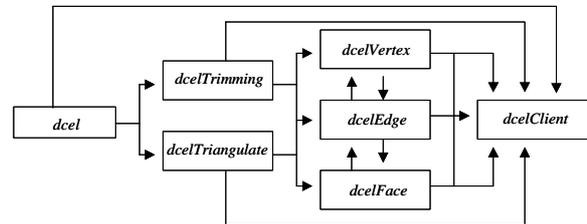


Figure 10. Communication flux between the classes of the library.

Methods of the *dcel* class perform the overall control of the algorithm. The *dcelTrimming* class is responsible for constructing trimming curves (Step II), while the *dcelTriangulate* class is responsible for rebuilding the finite-element mesh (Step III).

The *dcelClient* class is very important in this organization, because it is in charge of the communication between the library and the modeler. It describes the generic methods that should be overloaded by the modeler. This is the only task that the modeler programmer should perform to incorporate the class library into the modeler. The generic methods are described as follows:

- *getParametricMesh*: this method gets an input surface mesh that is converted into the topological data structure used by the algorithm. It is called in the beginning of the surface-intersection process.
- *getConstraint*: this method gets surface-mesh constraints; for example, edges on intersection curves obtained in a previous step. It is called in the beginning of the surface-intersection process.
- *evalSurface*: given a parametric coordinate, this method computes the respective 3D coordinates and their partial derivatives. It is called during the whole surface-intersection process.
- *closestSurfacePoint*: given a surface 3D point, this method determines the correspondent parametric coordinates. It is called during the whole surface-intersection process.
- *getTrimmingCurve*: given a set of interpolation points defined by the algorithm along a trimming curve, this method gets a reference of a parametric representation of this curve defined by the modeler.
- *getCurveSub*: given a (trimming) curve and a characteristic size, this method computes equally-spaced points along the curve. This characteristic size is defined by an average value of the sizes of intercepted edges in the input meshes.
- *newPatchMeshes*: for each input surface, this method passes the modeler a set of patch meshes resulting from the intersection algorithm.

5. APPLICATION EXAMPLES

To validate the ideas presented and to verify the robustness and efficiency of the extended algorithm described in this paper, the class library was incorporated to the MG modeler [8,13], increasing its ability of modeling complex Engineering problems. To illustrate its new capabilities, this section presents some modeling examples.

The first example is the modeling of two cylinders, one intercepting the other. Figure 11 shows the original surface meshes, while Figure 12 presents the meshes resulting from surface intersection. In Figure 13, some surface patches resulting from the intersection were removed, showing MG's ability to generate models from the composition of several modeling components.

Figures 14 to 16 illustrate the intersection of a torus model with a cylinder model. Figure 16 shows the complex intersection curve obtained by the algorithm.

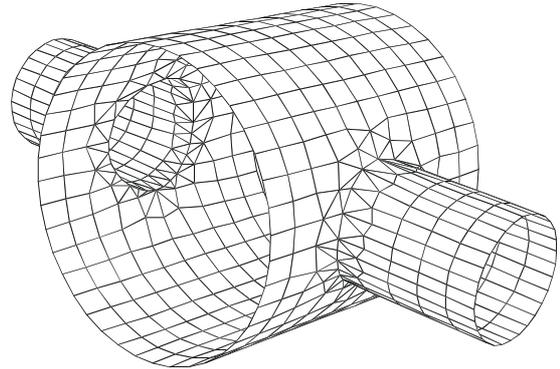


Figure 13. Final mesh of two-cylinder model after removing some patches.

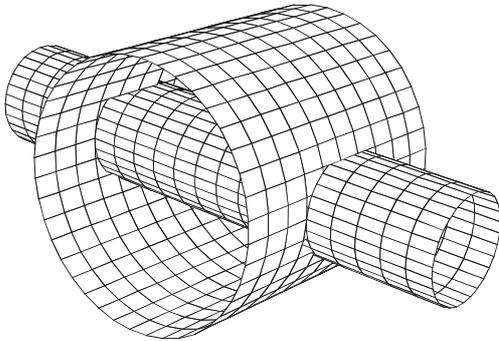


Figure 11. Original surface meshes of two cylinders that intercept each other.

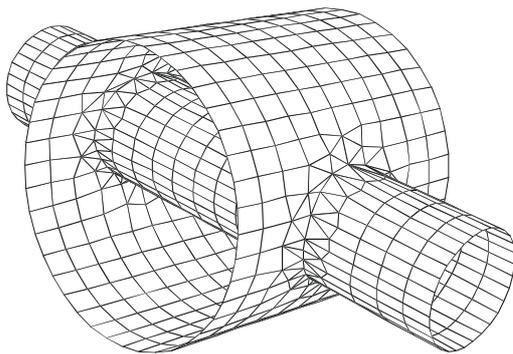


Figure 12. Meshes resulting from the surface intersection of two cylinders.

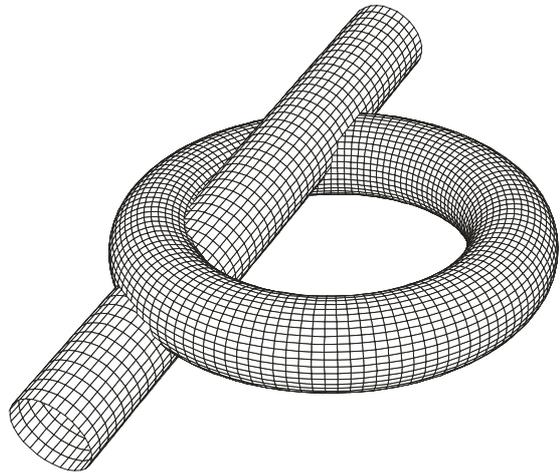


Figure 14. Torus and cylinder meshes before surface intersection.

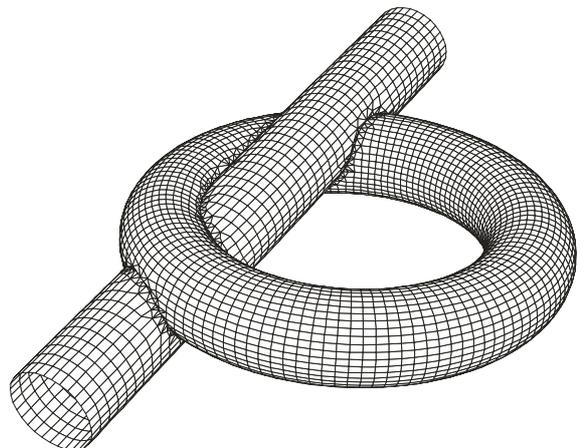


Figure 15. Mesh after intersection of torus and cylinder models.

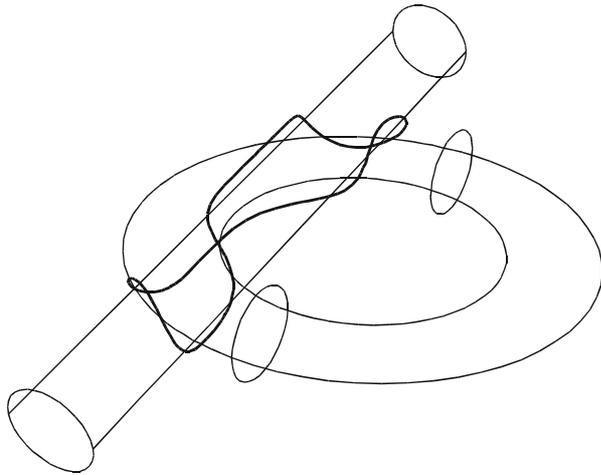


Figure 16. Curve resulting from torus and cylinder intersection.

The final example was created to demonstrate the ability of the present intersection algorithm to treat singular points resulting from the intersection of two surfaces. Figure 17 shows the original meshes of two identical cylinder surface patches with a relative rotation about a vertical axis. Figure 18 depicts the final mesh after intersection and Figure 19 illustrates the surface-patch boundaries and the intersection curves.

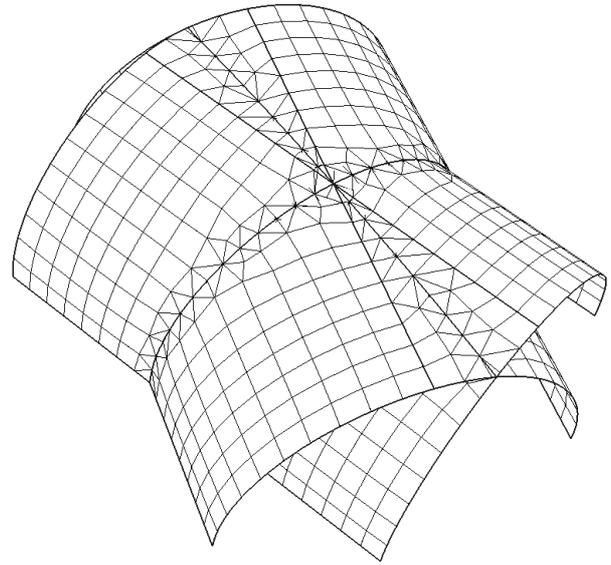


Figure 18. Two-cylinder model: mesh resulting from surface intersection.

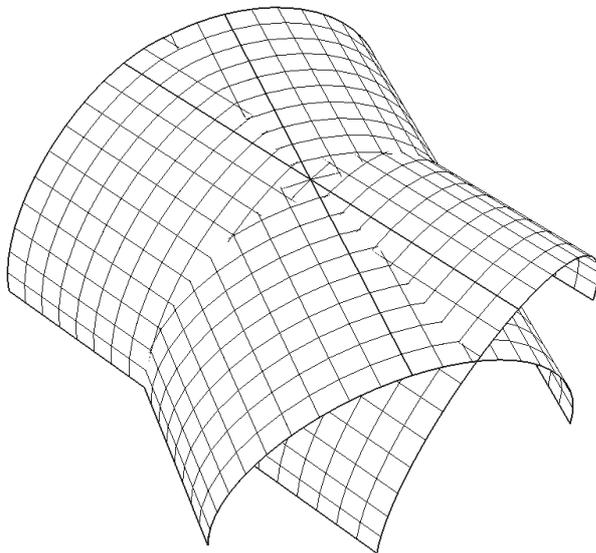


Figure 17. Two-cylinder model: surfaces before intersection.

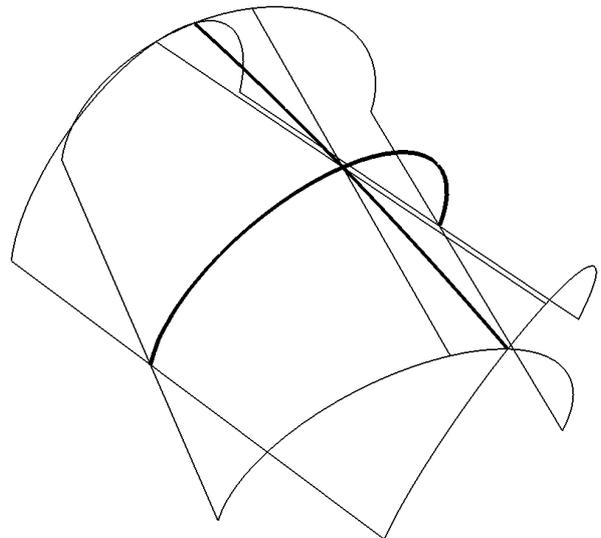


Figure 19. Two-cylinder model: surface-patch boundaries and intersection curves.

6. CONCLUSION

This work has described procedures to compute multiple intersections of finite-element surface meshes. These procedures are based on an extended version of a previously proposed surface-mesh intersection algorithm [8], treating special cases that had not been considered in

its original version, specially the intersections of more than two surfaces.

This work has also presented a class organization, in the context of OOP, of the data structure used in the surface-intersection algorithm, allowing its incorporation by any geometric modeler. This class organization also allows the generic handling of several types of surface geometry.

The integration of the presented class organization with any modeler is relatively simple. For such, it is only necessary to re-write generic methods defined in the class organization. To verify the validity of the proposal here presented, the class organization was incorporated to the MG modeler, increasing its capabilities of modeling surfaces with multiple intersections. Some examples using MG were shown to illustrate these capabilities.

The new elements generated by the intersections are very well shaped after the smoothing process of the original algorithm. An idea to further reduce the number of sliver facets that can still remain at the boundaries of the patches is to move the vertices along the curves, respecting only corners points. However, this has not been implemented yet.

ACKNOWLEDGEMENTS

The first author acknowledges a doctoral fellowship provided by CNPq. The third author acknowledges financial support by CNPq (Project 300.483/90-2). The authors acknowledge the financial support provided by agency Finep/CTPetro, process numbers 65999045400, 650003600 and 2101033200. The authors are grateful to Carolina Alfaro for the manuscript copydesk. The present work has been developed in Tecgraf/PUC-Rio (Computer Graphics Technology Group).

REFERENCES

- [1] C.M. Hoffmann, "Geometric & Solid Modeling: An Introduction", Morgan Kaufmann Publishers, 1989.
- [2] O.C. Zienkiewicz and R.L. Taylor, "The Finite Element Method", Fifth ed., Vols. 1 and 2, Butterworth-Heinemann, 2000.
- [3] R.E. Barnhill, G. Farin, M. Jordan, and B.R. Piper, "Surface/surface Intersection", *Computer Aided Geometric Design*, Vol 4 pp 3-16 (1987).
- [4] R. Barnhill and S. Kersey, "A Marching Method for Parametric Surface/surface Intersection", *Computer Aided Geometric Design*, Vol 7 pp.257-280 (1990).
- [5] Tz.E. Stoyanov, "Marching Along Surface/surface Intersection Curves with an Adaptive Step Length", *Computer Aided Geometric Design*, Vol 9 pp.485-489 (1992).
- [6] E. Houghon, E. Emmett, R. Factor, and L. Sabharwal, "Implementation of a Divide-and-Conquer-Method for the Intersection of Parametric Meshes", *Computer Aided Geometric Design*, Vol 2 pp.173-184 (1985).
- [7] S. H. Lo, "Automatic Mesh Generation over Intersecting Surfaces", *International Journal for Numerical Methods in Engineering*, Vol 38 pp.943-954 (1995).
- [8] L.C.G. Coelho, M. Gattass and L.H. Figueiredo, "Intersecting and Trimming Parametric Meshes on Finite-Element Shells", *International Journal for Numerical Methods in Engineering*, Vol 47 pp.777-800 (2000).
- [9] ACIS 3D Geometric Modeler, <http://www.spatial.com/products/3D/modeling/ACIS.html>.
- [10] Parasolid - Powering the Digital Enterprise, <http://www.plmsolutions-eds.com/products/parasolid/>.
- [11] Open CASCADE, <http://www.opencascade.com/>.
- [12] Pro/ENGINEER API Toolkit, http://www.ptc.com/products/proe/app_toolkit.htm.
- [13] W.W.M. Lira, P.R. Cavalcanti, L.C.G. Coelho, and L.F. Martha, "A Modeling Methodology for Finite Element Mesh Generation of Multi-Region Models with Parametric Surfaces", *Computer & Graphics*, Vol. 26(6), in press (2002).
- [14] F.P. Preparata and M.I. Shamos, "Computational Geometry – An Introduction", Springer Verlag, New York, 2000.
- [15] D. Comer, "The Ubiquitous B-tree", *ACM Computing Surveys*, Vol 11 pp.121-131 (1979).
- [16] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp.322-332 (1995).
- [17] M.R. Mediano, M. Gattass, and M.A. Casanova, "HPS-tree: An access method for storing long maps with geometrix and topologic multi-resolution", *SIBGRAP'1996 – IX Brazilian Symposium on Computer Graphics and Image Processing*, pp.219-226 (1996).