INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING, VOL. 0(0), 1–100 (1999)

INTERSECTING AND TRIMMING PARAMETRIC MESHES ON FINITE-ELEMENT SHELLS

LUIZ CRISTOVÃO G. COELHO   MARCELO GATTASS*
Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil

LUIZ HENRIQUE DE FIGUEIREDO
LNCC–Laboratório Nacional de Computação Científica, Petrópolis, Brazil

ABSTRACT

We present an algorithm for intersecting finite-element meshes defined on parametric surface patches. The intersection curves are modeled precisely and both meshes are adjusted to the newly formed borders, without unwanted reparametrizations. The algorithm is part of an interactive shell modeling program that has been used in the design of large offshore oil structures. To achieve good interactive response, we represent meshes with a topological data structure that stores its entities in spatial indexing trees instead of linear lists. These trees speed up the intersection computations required to determine points of the trimming curves; moreover, when combined with the topological information, they allow remeshing using only local queries.

KEY WORDS: surface intersection; finite-element mesh generation; parametric representation; geometric modeling; topological data structures; constrained triangulation

1. INTRODUCTION

Surface modeling and mesh generation on surfaces are important problems in computer-aided geometric design (CAGD), specially in engineering applications that use the finite-element method for analyzing shell structures. One way to build complex models is to take several simple surface patches and sew them together into a single surface, trimming excess parts away. If this modeling technique is to be effective, then the problem of computing the intersection between two patches must be solved efficiently and robustly.

We consider a modeling environment in which surface patches are interpolated from boundary curves given in parametric form. Such surfaces are called *Coons patches*.[1] The user provides discretizations for the boundary curves, which are then used to interpolate a mesh for the surface.

We approach the surface intersection problem as a *meshing* problem, in the sense that we are given not only two parametric surface patches, but also meshes on them. Both the parametrizations and the meshes must be taken into account when computing the intersection. In other words, we must compute intersection curves that lie on both surfaces *and* on both meshes. Moreover, the boundary discretizations provided by the user must be respected, so that adjacent meshes fit together correctly. Thus, our goal is to compute a single mesh for the union of the two patches, which must be composed largely

* Correspondence to: Prof. Marcelo Gattass, Department of Computer Science, PUC-Rio, Rua Marquês de São Vicente 225, 22453-900, Rio de Janeiro, RJ, Brazil. E-mail: gattass@inf.puc-rio.br.

CCC 0029–5981/99/000001–100

Received September 29, 1998
Revised June 15, 1999

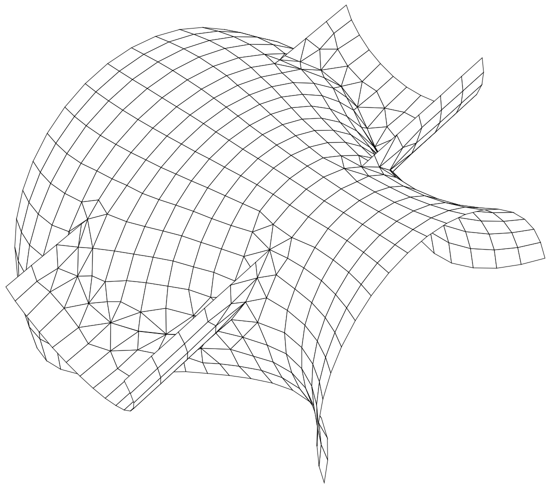


Figure 1. Composite mesh after intersection.

of the original meshes, modified only locally, around the intersection. Figure 1 shows an example of a composite mesh.

In this paper, we present a fully automatic algorithm that solves this mesh intersection problem. The searches required for computing the intersection curves and for re-meshing are supported by a topological data structure whose main feature is that topological entities are stored in B-trees[2] and R*-trees,[3] instead of linear lists. These spatial indexing structures play a major role in the overall efficiency of the algorithm.

The algorithm has been implemented and is part of a complete shell modeling program, called MG,[4] which has been used in the design of several large offshore oil structures at PETROBRAS, the Brazilian Oil Company. Interactive shell modeling by direct manipulation in 3D space raises several interesting user-interface issues that have been discussed elsewhere.[5]

Section 2 reviews previous related work, highlighting some of the problems that arise in simple intersecting procedures. The requirements that a good algorithm to solve the mesh intersection problem should satisfy are discussed in Section 3. The algorithm described in detail in Section 4 aims at fulfilling these requirements. Section 5 shows examples of intersections between pairs of patches, and Section 6 shows examples of composite surfaces with several intersections. Finally, Section 7 contains conclusions and lists directions for future work.

2. PREVIOUS WORK

Parametric mappings provide a simple and effective method for generating finite-element meshes on general curved surface patches given by boundary curves. Haber and Abel[6] pioneered the use of graphical interfaces with transfinite mappings to generate meshes in shell structures. Their work was followed by a few others,[7,8] who introduced new techniques for constructing meshes on trimmed patches. However, these works do not focus on the surface intersection problem, which is a preprocessing stage for constructing meshes on surfaces.

Only rarely the intersection of two parametric surfaces can be computed analytically. Even when it can, an analytical solution might not be practical. Hence, we must resort to numerical techniques.

The methods for solving the surface intersection problem in CAGD belong to two major classes: marching methods and decomposition methods. *Marching* or *continuation methods* compute the intersection curve in 3D object space by marching along the curve in the direction of its tangent vector.[9–12] *Decomposition* or *subdivision methods* compute the trimming curves in 2D parameter space by recursively refining the solution at each step.[13–15]

Previous solutions to the surface intersection problem work well in many cases, but do not handle the mesh intersection problem as defined in Section 1. An exception is the work of Lo,[16] which motivated the present work.

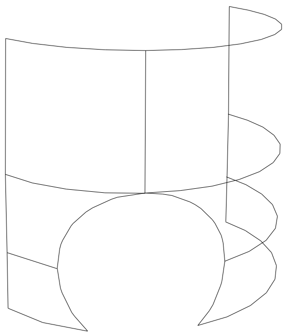
Lo[16] proposed a simple algorithm for intersecting triangular meshes that automatically redefines the meshes to adapt to the intersection curves. Lo's solution does not use the continuous parametric representation of the surfaces, and thus can be used easily in most modeling systems. On the other hand, around regions of high curvature, the computed intersection points may not lie on the original surfaces, which may yield unacceptable results in many shell problems.

3. ALGORITHM REQUIREMENTS FOR MESH INTERSECTION

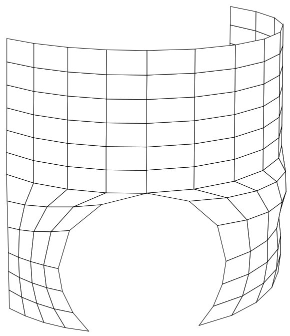
The well-known requirements for good solutions for the surface intersection problem in CAGD [17, p. 505] must be refined for the mesh intersection problem. We believe that an ideal solution for the mesh intersection problem should satisfy the following requirements:

Correctness. The computed intersection curves must lie on both surfaces. If the intersection curves lie only on the faces of the meshes but not on the surfaces, then the finite-element analysis may yield false or unacceptable results. The easiest way to guarantee correctness is to compute the trimming curves in parametric space and then map them to object space.

Faithfulness. The geometry of the resulting surface must faithfully reflect the geometry of the original surfaces, because they represent the intent of the designer. In particular, new parametric patches should not be defined using the trimming curves as boundary curves, because this would yield a different geometry. For example, the bulge in the cylindrical shell shown in Figure 2 is due to a Coons subpatch that contains part of the circle as a boundary; this subpatch is not cylindrical.



(a) Boundary curves for the subpatches after trimming.



(b) Bulge alters original geometry.

Figure 2. Bulge on cylindrical shell produced by reparametrization.

Precision. Points in parametric space corresponding to the same intersection point, when mapped to object space, must be at a distance smaller than a tolerance selected by the user. Usually, this tolerance is a fraction of the size of the smallest intersected element in the original meshes.

Self control. The two meshes should be redefined automatically to include the intersection curves and possibly exclude trimmed regions. The user should not be required to edit the resulting meshes manually.

Quality. The elements generated during re-meshing must have good geometric shape and must be of the same average size as the elements in the original meshes. When the original meshes have elements with very different sizes, the resulting mesh may have elongated elements with sharp angular differences, if no corrections are made. A common technique for ensuring mesh quality is to run a post-processing smoothing step. If smoothing is performed, then it should be automatic, with no user intervention.

Locality. Only elements close to the intersection region should be modified during re-meshing. Elements far from the intersection region should remain unaltered. If smoothing is used to improve mesh quality, then its effect should be local too.

Regions. The regions defined in parametric space by the trimming curves must be identified automatically. These regions may or may not take part in the final model; they can even have different attributes (loads, materials, boundary conditions).

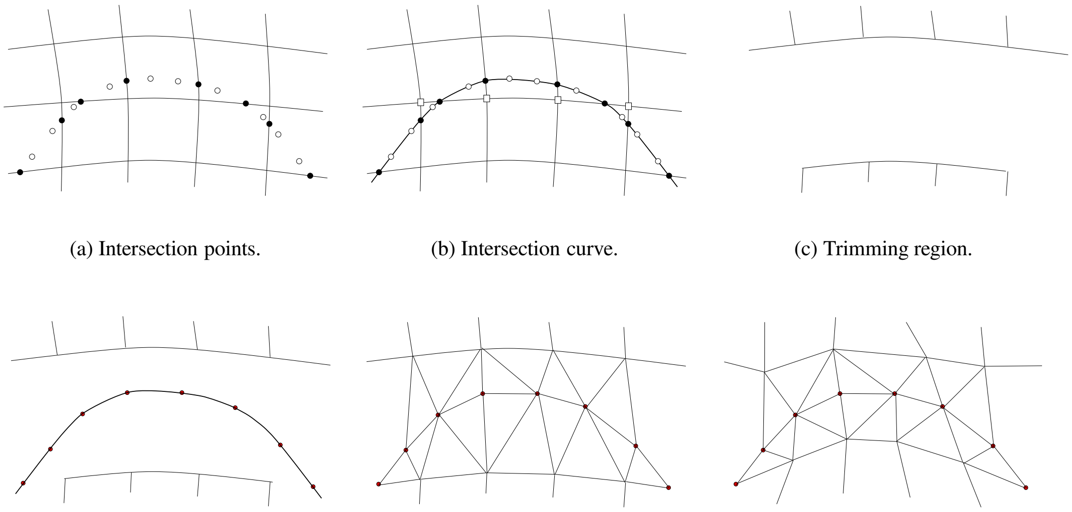
Efficiency. The time and space required for computing the intersection should be ideally linear in the number of elements around the intersection region. Algorithms that are quadratic in the total number of elements in the meshes are too slow for large meshes and are not appropriate for interactive modeling.

Robustness. An arbitrary number of intersection curves may be generated, having different geometry, topology, and interpolation points. In particular, closed curves must be handled correctly, and must yield regions with holes when trimmed.

4. THE ALGORITHM

The algorithm we propose here for intersecting two meshed surfaces A and B has three main steps, which aim at fulfilling the requirements listed in Section 3. Figure 3 illustrates these steps.

1. Determination of the intersection points:
   (a) Compute and store the intersection of edges of A against faces of B; and
   (b) Compute and store the intersection of edges of B against faces of A.
2. Determination of the trimming curves:
   (a) Link the intersection points found in Step 1 into a set of polygonal lines that represent the trimming curves;
   (b) Interpolate parametric curves through these points;
   (c) Compute uniformly distributed sample points on these parametric curves; and
   (d) Move these new points onto each surface.
3. Remeshing:
   (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 2;
   (c) Triangulate the trimming regions on both surfaces; and
   (d) Smooth both meshes.



(a) Intersection points. (b) Intersection curve. (c) Trimming region.

(d) Insertion of trimming edges. (e) Triangulation of trimming region. (f) Mesh after smoothing.

Figure 3. An overview of the algorithm.

Figure 3a shows the intersection points computed in Step 1. The filled circles mark the positions where the edges of A cross the faces of B; the empty circles mark the positions where the edges of B cross the faces of A. To avoid testing all edges of one surface against all faces of the other, we store topological entities in spatial indexing trees, as described in Section 4.1. It is here that these trees contribute most to the efficiency of the algorithm.

Since edges and faces are curved in object space, we need a numerical procedure to determine the intersection points, which is described in Section 4.2. At the end of Step 1, edges in one mesh are paired with the faces that they intersect in the other mesh, and vice-versa. For each such pair, we also store the parametric coordinates of the intersection point in both domains.

Figure 3b shows the trimming curve obtained in Step 2 from the intersection points computed in Step 1. As described in Section 4.3, we first link the intersection points computed in Step 1 into a set of polygonal lines, which represent the trimming curves in parametric space. Only topological tests are necessary to link the intersection points — this avoids expensive geometric searching, used in previous approaches.[16,18] Next, we convert these polygonal lines into continuous curves, which are then uniformly sampled. Finally, these samples points are relaxed back onto the original surfaces,

Figure 3c illustrates Step 3a, in which the trimming region is found by removing the edges adjacent to the vertices marked with squares in Figure 3b. Figure 3d shows the new edges that connect equally-spaced points sampled on the continuous parametric trimming curve computed in Step 3b. Figure 3e shows the triangulation computed in Step 3c: each trimming region is triangulated by inserting edges according to the geometric criteria described in Section 4.4. Finally, to increase mesh quality, we use a hybrid smoothing technique, done partially in parametric space and partially in 3D, as described in Section 4.5. Figure 3f shows the final, smoothed mesh (Step 3d).

4.1. The data structure

We store the meshes in a variant of the DCEL topological data structure,[19] extended to link two planar subdivisions and to handle surfaces, as shown in Figure 4. Moreover, topological entities (vertices, edges, and faces) are stored in spatial indexing trees instead of the usual linked lists or arrays. This allows fast solutions for the searches necessary for the initial mesh assembly and for remeshing.

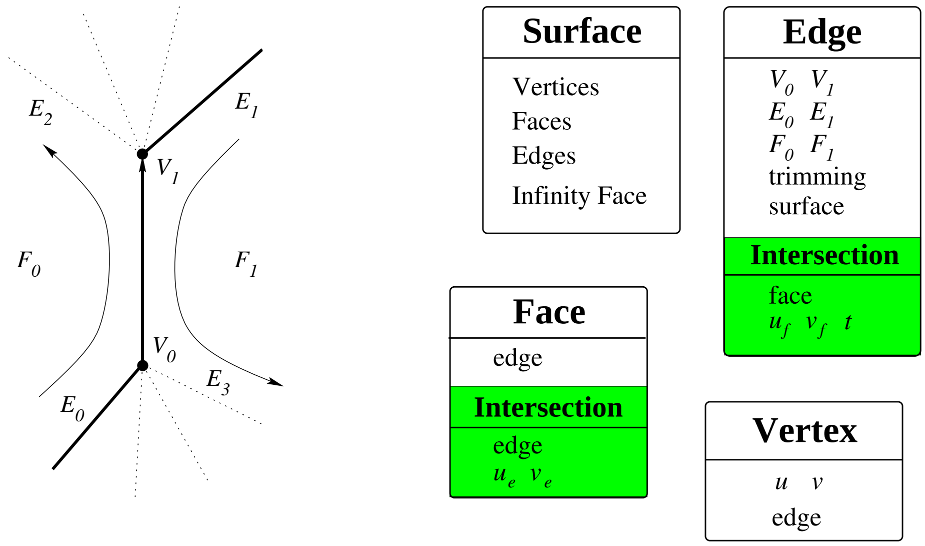
We use B-trees[2] for storing vertices and edges, and R*-trees[3] for storing faces. The vertices are inserted into a B-tree that searches points by their parametric coordinates in lexicographical order. Each vertex gets an index such that vertices with smaller indices have smaller parametric coordinates (in the lexicographical sense).

The edges, which are considered straight in parametric space, connect pairs of vertices and are oriented from the vertex with the smaller index to the vertex with the larger index. The edges are stored into a B-tree that uses these indices as the search key, as in the vertex B-tree.

The faces are inserted into an R*-tree that uses their 3D bounding boxes as the search key. We use R*-trees with nodes with a minimum of four and a maximum of ten faces because they present good average performance for planar subdivisions.[20]

As shown in Figure 4b, each surface has a well-defined two-dimensional parametrization (Coons patches, Bézier surfaces, NURBS, etc.), references to the vertex, edge and face trees, and a single reference to the *infinity face*.

The Intersection field is used to determine the trimming curves and for mesh reconstruction; it is also the key to the topological chaining of two DCELs. Figure 5 shows how this topological chaining between the faces and edges of two intersecting surfaces is made. The parametric coordinates of an intersection point relative to each surface is stored in (u_f, v_f) and (u_e, v_e).



(a) Edge adjacencies. (b) Topological entities.

Figure 4. The modified DCEL.

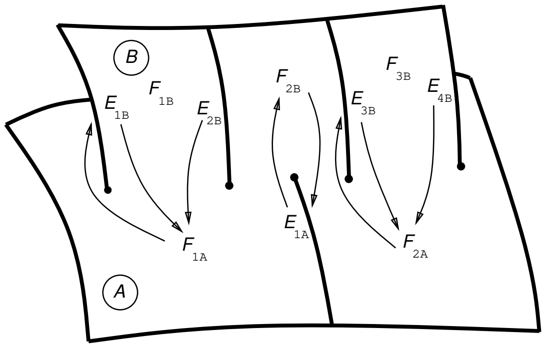


Figure 5. Topological chaining of DCELs.

The set of Euler operators that manage the DCEL used here is described in Appendix A. Care has been taken in the implementation of these operators to perform only local queries with the entities involved with the changes. The searches are performed before the calls to the operators, using the corresponding key to the vertex, edge or face tree. In Appendix B, we show how the construction of the initial mappings is done with the described Euler operators.

4.2. Computation of intersection points

In Step 1 of the algorithm, each edge in one mesh is tested against the faces in the other mesh that can potentially intersect it. This is done by testing their bounding boxes for intersection. The face R*-tree is searched by using the edge bounding box as key. Figure 6 shows the intersection of a straight edge against a Coons patch. The long box contains the edge; the small boxes are the bounding boxes of the faces in the patch that potentially intersect the edge.

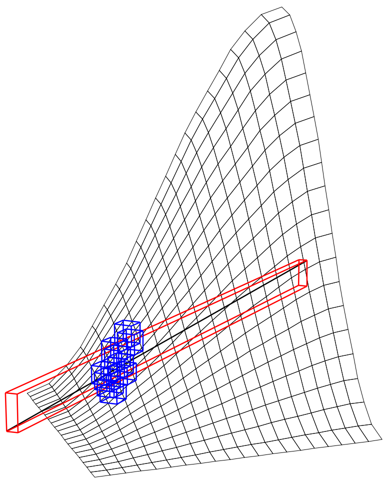


Figure 6. Potential intersection between an edge and a Coons patch.

For each selected edge-face pair, we find the point where they intersect (if any) by solving a system of non-linear equations: $F(u,v,t) = 0$, for $F(u,v,t) = S(u,v) - R(t)$, where $S(u,v)$ is a parametrization for the face, and $R(t)$ is a parametrization for the edge. We use a modified Newton-Raphson method[21] to solve this system; the algorithm stops when the distance from the edge point $R(t)$ to the face point $S(u,v)$ is smaller than a given tolerance. This tolerance corresponds to a fraction of the smallest length of the edges involved in the intersection.

The methodology used here is similar to the one used by Chen and Ozsoy,[22] who regard the surface-surface intersection problem as a set of curve-surface problems. The *engaging direction* defined by Chen and Ozsoy[22] determines a straight line in parametric space connecting the current intersection point and the next one, based on the construction of the osculating circle. Considering the meshes formed by straight edges in parametric space, as we do, the edge-face problem is well defined and no auxiliary constructions are necessary. Figure 7 illustrates the elements involved in the problem, where the parameter t, which defines the variation throughout the intersected edge, can be expressed as an interpolation given by the straight line in parametric space that connects V_0 and V_1.

The correction term Δp in the Newton-Raphson method for solving $F(p) = 0$, where $p = (u,v,t)$, is the solution of the linear system $\mathbf{J}^t \Delta p = -F(p)$, where $\mathbf{J}$ is the Jacobian matrix of F. Since $\mathbf{J}$ is composed by the tangents to the surface and by the tangent to the curve, which is a directional derivative on the edge's surface, one must be very careful to ensure convergence. For instance, if the surfaces have different parametrizations, such as one bicubic and another bilinear, then a normalization of the tangent vectors in $\mathbf{J}$ is required before solving $\mathbf{J}^t \Delta p = -F(p)$. To make the procedure robust enough to cope with the singularities that occur where the surfaces are tangent or parallel to each other, we use the modified Newton-Raphson method described by Deuflard.[21]

The solution of $F(p) = 0$ gives two parametric pairs: (u_f, v_f) in the face's surface, and (u_e, v_e) in the edge's surface. The relative position of the face and the intersection point must be tested inside the face's parametric space. A standard 2D point-in-polygon algorithm[23] checks whether the intersection point lies inside, outside, on a boundary, or on a vertex of the face being tested.

Figure 8 shows the relative positions that an intersection point between an edge of surface A (only the edge is drawn) and a face of surface B (shown in dark gray) can assume. The faces in surface B that must have their Intersection fields updated are shown in light gray. When the point is outside the face, no updates have to be made in the data structure (Figure 8a). When the point is inside the face, the edge and face must be updated (Figure 8b). When the point is on an edge, the intercepted edge in surface A and the two adjacent faces in surface B must also be updated (Figure 8c). When the intersection is located on a vertex of the face, all faces adjacent to this vertex must be updated (Figure 8d).

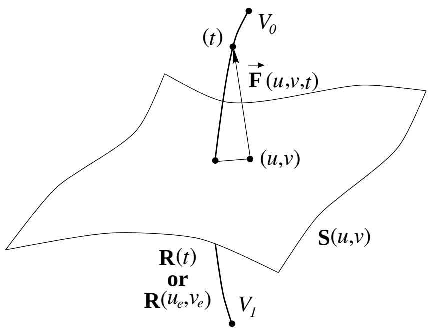
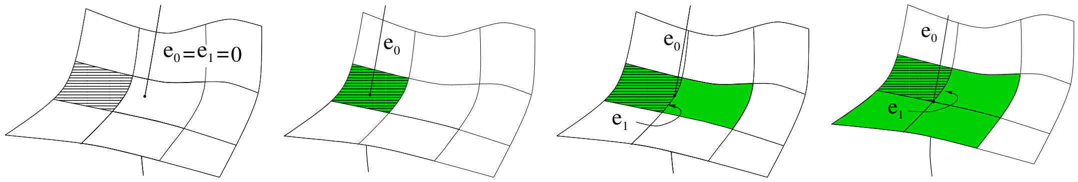


Figure 7. Notation used for computing intersection points.



(a) Point outside face. (b) Point inside face. (c) Point on edge. (d) Point on vertex.

Figure 8. Relative positions of the intersection point and a face.

If a new edge intersects a face that has already been intersected by another edge, then the face's Intersection field is not updated with this new edge, because, from a reference to any one intersecting edge, all other edges can be found using the topological representation of the mesh.

At the end of Step 1, all topological and geometrical information necessary to determine the trimming curves is stored into the DCELs.

4.3. Computation of intersection curves

In Step 2, the algorithm first visits each face of one surface to determine the segment of the intersection curve inside it. This segment is represented by a polygonal line, called a *local polyline*, which links the points where edges of the other surface intersect the face, and the points where the boundary of this face intersects the other surface.

After finding the local polyline on the initial face, the algorithm moves to an adjacent face, computes its local polyline, and then joins the two local polylines into a *global polyline*. The algorithm proceeds in this way until all adjacent faces have been visited, and a global polyline has been completed.

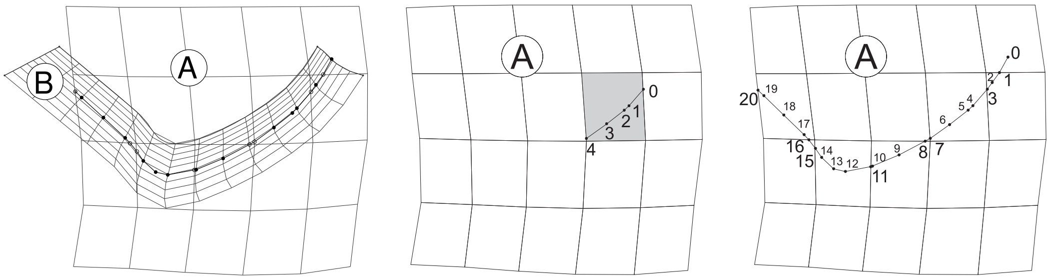
In the general case, the intersection of two patches may be formed by several separate curves. In this case, Step 2a finds a global polyline for each component: after finding the first global polyline, the algorithm searches for a new face whose Intersection is marked and that has not already been visited. It then proceeds as before, and builds the corresponding global polyline.

Step 2a stops when all intersected faces in both meshes have been visited. Thus, all connected components of the intersection are found with a single search through the intersected faces. To take advantage of the face R*-tree, the algorithm actually computes the bounding box of the intersection points found in Step 1 and, using spatial indexing, extracts only the faces that intersect this box.

Local and global polylines are simply lists of points. Each point contains two references for edges: e_0 and e_1. For regular intersection points, as shown in Figure 8b, only field e_0 is used; it contains a reference to an edge in one of the intersecting surfaces. Field e_1 is used only when the intersection point lies on the intersection of two edges, as shown in Figures 8c and 8d. In these cases, e_1 must point to an edge on the other surface different from the one in e_0. In the case shown in Figure 8d, any edge adjacent to the intercepted vertex may be stored in e_0.

Figure 9a shows the intersection curve and the intersection points of two surfaces A and B. Figure 9b shows the local polyline for the gray face. Points 0 and 4 are the intersections of the edges of this face with faces of B; points 1, 2, and 3 are the intersections of edges of B in the interior of the gray face. Figure 9c shows the global polyline obtained by propagating local polylines through all faces of A.

Regarding the position of the trimming curve and a given face, our algorithm handles the three cases



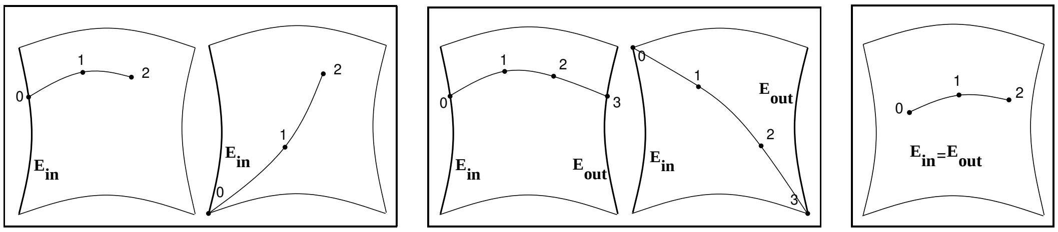
(a) Intersection points. (b) Local polyline. (c) Global polyline.

Figure 9. Local and global polylines used in Step 2 of the algorithm.

shown in Figure 10: (a) *singlecrossed*, (b) *doublecrossed*, and (c) *isolated* or with no edge crossing. The most common case is *doublecrossed*, where the trimming curve crosses the face. Note that the global polyline is always propagated over the intersected edges, and it can only end at an intersection point that is inside a face or on an edge (vertex).

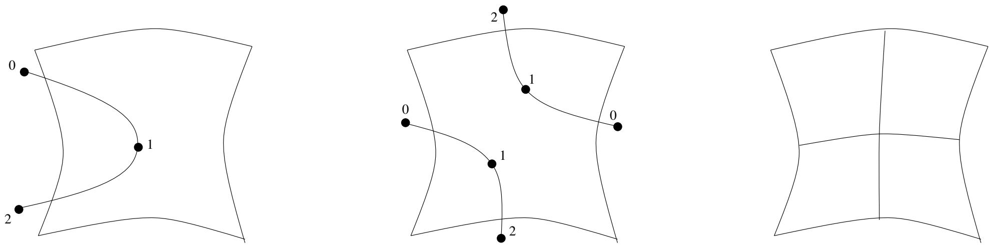
When the edge intersections occur in one of the vertices (points 0 and 3 on the right-hand side of Figures 10a and 10b), the global polyline is propagated to all faces adjacent to this vertex. These cases are identified in Step 1, when the Intersection fields of all adjacent edges are updated. Given the uniform nature of finite-element meshes, edge intersections are very likely to occur, which is confirmed in the examples shown in Section 5.

In general, two parametric patches may intersect in more ways than the ones shown in Figure 10. The intersection can be a surface (when both patches concur), one or more curves, an isolated point, or any combination of these. Our goal is not to build an algorithm that detects all geometric possibilities of intersections between two patches, since we are concerned with remeshing. Instances of coincident patches and isolated points resulting from mutually tangent patches are not treated here. We focus on the cases in which the intersections are well-defined curves and the patches have a mesh compatible with the complexity of the intersection curves. Edges that are crossed twice by the same intersection



(a) *Singlecrossed.* (b) *Doublecrossed.* (c) *Isolated.*

Figure 10. Classifications between face and local polyline.



(a) An edge crossed twice. (b) A face crossed by two different curves. (c) Solution.

Figure 11. Intersections cases not handled by algorithm (a) and (b) are avoided by using greater discretization (c).

curve (Figure 11a), identified in Step 1, and faces crossed by two intersection curves (Figure 11b), identified in Step 2, cannot be handled by the algorithm with the initial discretization. Both cases can be solved by increasing the mesh discretization, as shown in Figure 11c.

Figure 12 describes the algorithm that builds the list of global polylines. As mentioned before, the bounding box defined by the intersection points computed in Step 1 is used as the search key in the `for` loop in BuildGPolyList. Each intersected face that has not already been visited originates a global polyline, which is recursively propagated by algorithm BuildGPoly (shown in Figure 13).

BuildGPoly propagates each global polyline through the faces of a surface. The input for the algorithm are the face Fc in which the polyline starts and the polyline GPoly that will be propagated along the surface. When BuildGPoly ends, polyline GPoly has, for each intersection point, one (e_0) or two (e_0 and e_1) references to edges on the surfaces. This information is also used in the remeshing step (Step 3 of the main algorithm), without the need to perform searches to identify the relative position of the trimming curves and the faces and edges on both patches.

Note that recursion stops in two cases: when the face Fc has already been visited; or when Fc is the infinity face, because in this case GPoly has reached the boundary of the mesh. BuildLPoly creates the local polyline LPoly, which is a list of all intersections with Fc, and returns references to the possible intersected edges, Ein and Eout (Figure 10).

```
algorithm BuildGPolyList(S)
   input: surface S
   output: list of global polylines L
begin
   Initialize L as empty
   for each Face intersecting bounding box of all intersection points do
       if Face has Intersection and Face is not visited then
          Create new GPoly
          BuildGPoly(Face,GPoly)
          Add GPoly to L
       end if
   end for
end BuildGPolyList
```

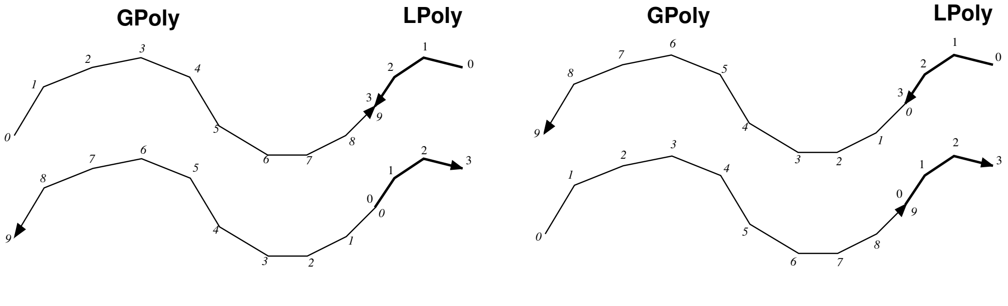
Figure 12. Algorithm for constructing the global polyline list.

```
algorithm BuildGPoly(GPoly, Fc)
   input: face Fc and global polyline GPoly
   output: global polyline GPoly completed
begin
   if Fc is marked or Fc is infinity then return
   Create local polyline LPoly
   BuildLPoly(LPoly, Fc, Ein, Eout)
   AddGPoly(LPoly, GPoly)
   Remove LPoly
   Mark Fc as visited
   for E=Ein, Eout do
      if intersection in E is over a vertex
         for each Fviz adjacent to the intersected vertex
            BuildGPoly(GPoly, Fviz)
         end for
      else
         Fviz ← face neighbor to E
         BuildGPoly(GPoly, Fviz)
      endif
   end for
end BuildGPoly
```

Figure 13. Algorithm for constructing global polylines.

The intersection points in LPoly are then inserted into one of the two extremities of GPoly by the algorithm AddGPoly (not shown). One of the fields (e_0 or e_1) present in the extreme points of LPoly must coincide with the initial or final point of GPoly, indicating the connection position. The direction of both polylines is also tested according to the adjacencies. Figure 14a shows the case in which the local polyline must be inserted in the inverted direction; Figure 14b shows the case in which the local polyline must be inserted in its natural order.

If Ein exists, then we test the value of the parameter t shown in Figure 4 to detect whether the intersection is over a vertex of Ein. If the intersection is at an intermediary position ($0 < t < 1$), then GPoly is propagated through the face Fviz adjacent to Fc by Ein. If the intersection is on a vertex of Ein ($t = 0$, indicating V_0; or $t = 1$, indicating V_1), then GPoly is propagated through all faces adjacent



(a) Inversion situations.

(b) Maintenance situations.

Figure 14. Relative directions of the local and global polylines.

to this vertex. If Eout exists, then the face is *doublecrossed*, and the tests and propagations procedures made with Ein are repeated for Eout. When no edge is intersected (Ein and Eout are null), GPoly is already completed and BuildGPoly stops.

Procedures BuildLPoly (not shown) and AddIntPoint (shown in Figure 15) use the topological information computed in Step 1 of the main algorithm to construct LPoly by alternately traversing both patches involved in the intersection. The BuildLPoly procedure loops over edges of face Fc to find the first one (Ein) that has an intersection, storing this edge at the first position of LPoly, if such edge exists. BuildLPoly continues by calling AddIntPoint, which inserts the intersection points internal to face Fc. If the first intersected edge Ein of face Fc exists, then another possible intersected edge Eout is searched for; if found, it is inserted at the last position of LPoly.

AddIntPoint starts from the other surface using the reference present in the field face of edge Ein. If Ein is `null`, then no topological information is available and the algorithm searches the R*-tree to find the face that has an edge whose Intersection field points to Fc. Note, however, that this search is necessary only in the cases of isolated polylines, as shown in Figure 10c. In the `while` loop of the algorithm, the edge Eint of Fint that has intersection with Fc is determined, followed by the storage of this intersection in LPoly, and the assignment of the adjacent face from Fint trough Eint to Fint. This loop stops in one of the three cases: when the adjacent face has already been visited, when it corresponds to the infinity face, or when there is no other intersected edge with face Fc.

To convert the global polylines found in Step 2a into continuous intersection curves, we interpolate the computed intersection points using cubic splines. We use the knot sequence selection scheme of Foley and Nielson[24] because it gives good curvature profiles for points having non-uniform spacing (it also has other interesting properties, such as invariance under affine transformations).

After interpolation, we sample the trimming curves uniformly, with a spacing proportional to the size of the edges in the initial mesh. The sample points define vertices on the intersection curves in the combined mesh. Although the interpolation points lie on both surfaces, the new sample points may not, and we have to relax them back onto the surfaces, using a procedure similar to the one used in Step 1. The convergence is much faster in this case, due to the proximity of the starting positions. As a by-product, we also obtain the parameter values for the sample points.

```
algorithm AddIntPoint(Fc,Ein,LPoly)
   input: face Fc, edge Ein and local polyline LPoly
   output: polyline LPoly with the intersections internal to Fc
begin
   if Ein is not null then
     Fint ← Ein->face
   else
     GetFaceIntersectingFace(Fc,Fint)       (this uses the R$^*$-tree)
   end if
   while Fint is not null, not the Infinity, and not visited do
     GetEdgeIntersectingFace(Fint,Eint)
     if Eint is not null then
       add Eint to LPoly
       Fint ← the face adjacent to Fint by Eint
     else
       Fint ← null
     end if
   end while
end AddIntPoint
```

Figure 15. Insertion of the intersections internal to a face.

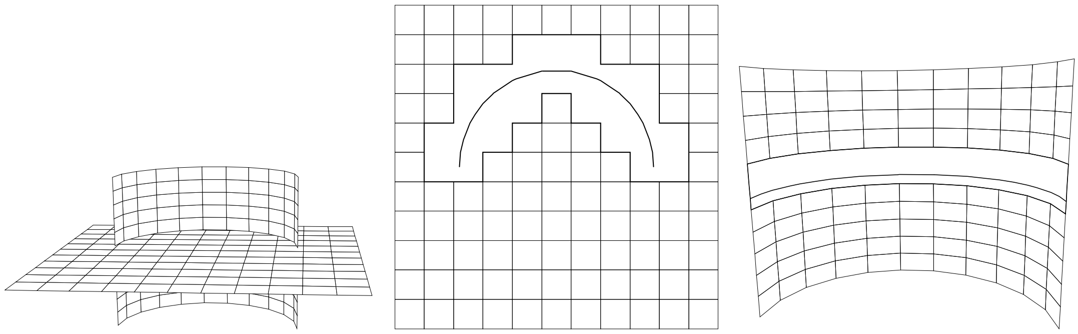
4.4. Remeshing

Each curve generated in Step 2 originates a trimming region on each patch by the removal of some edges. Each trimming region is actually a face of the DCEL, which is enlarged as the edges adjacent to vertices close to the intersection curves are eliminated. The intersected edges are stored directly into the fields e_0 and e_1 of each interpolation point of the intersection curves. To obtain the trimming regions in Step 3a, for each interpolation point in the trimming curves, we remove all edges connected to the vertex closest to this point (marked with squares in Figure 3b), including the intersected edge itself. The determination of the closest vertex is done by evaluating the edge parameter t, which indicates the intersection position: if t is smaller than 0.5, then we remove all edges connected to V_0; if t is greater than 0.5, then we remove all edges connected to V_1.

The intersected edges belonging to a boundary (identified as being adjacent to the Infinity face) and those created over trimming curves computed previously (identified by the trimming field) cannot be removed, and are split by inserting a new vertex at the crossing position. Figure 16 illustrates the trimming regions for the intersection of a plane and a cylindrical shell.

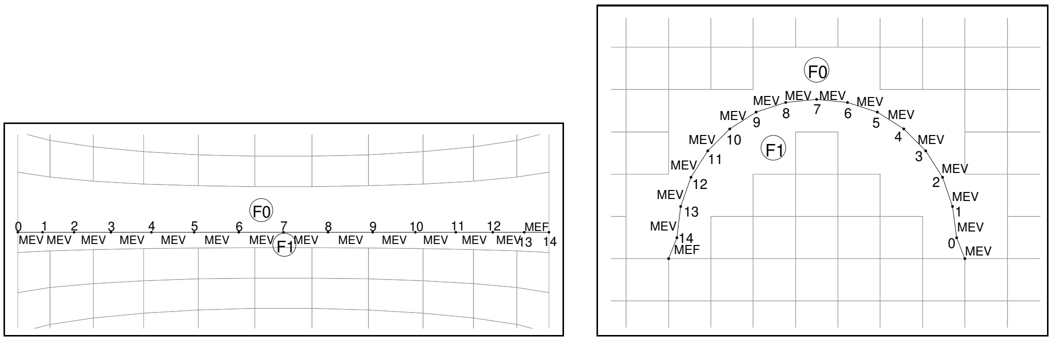
The trimming region or face has only one cycle of vertices when the trimming curve is opened. If the edge removal technique were applied to the closed trimming curves without restrictions, self-connected faces with an external and an internal cycle would be created, and this could not be handled by the DCEL. For this reason, the closed trimming curves are treated in a somewhat special way, by splitting some intersected edges, as will be presented next with the insertion of trimming edges.

In Step 3b, the edges representing the trimming curves are inserted, dividing the faces in two, one to the left and one to the right of each curve. The Euler operators described in Appendix A are used to insert the trimming edges. The edges created over the trimming curves receive the positive mark on the trimming field. Each extreme of the curve may or may not belong to a boundary edge. Figure 17a shows the Euler operators used to insert the edges that will represent the curve in the patch representation, subdividing the trimming face of Figure 16c. The boundary points (0 and 14) were already inserted with the split of the corresponding boundary edges during Step 3a. The following segments of the curve's subdivision are inserted with the MEV operator, except for the last one (points 13 to 14), created by the MEF operator which divides the trimming face in two (faces $F0$ and $F1$ in Figure 17a).



(a) Visual intersection. (b) Trimming region on plane. (c) Trimming region on cylinder.

Figure 16. Trimming regions in a plane $\times$ cylindrical shell intersection.



(a) Cylindrical shell.

(b) Plane.

Figure 17. Operators used for inserting trimming edges in the faces of Figure 16.

Figure 17b shows how the plane trimming face is split, and how the extreme points are connected to the trimming face. An initial MEV operator connects the first trimming curve point (0) to the closest vertex of the trimming face; then, a sequence of MEVs done for every segment of the trimming curve is made. Finally, a MEF connecting the last point (14) to the closest point in the trimming face completes the face splitting, generating the new face $F1$.

To deal with the case of closed intersection curves, the first position of the curve is inserted with the splitting of the edge stored in it, in spite of being removed as an internal edge, thus avoiding the creation of a face with two vertex loops in Step 3b. The middle position of the intersection curve is also connected to both sides of the trimming face to create four trimming faces on each patch. Figure 18 shows an example of a closed trimming curve over a cylindrical shell, and the operators used for inserting the trimming curve and the trimming faces $F0$ through $F4$.

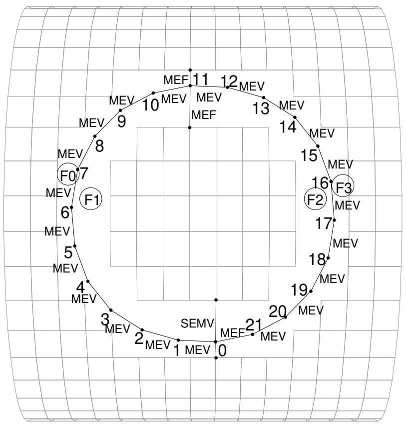
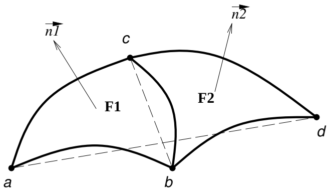
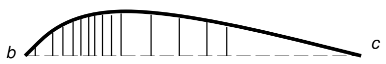


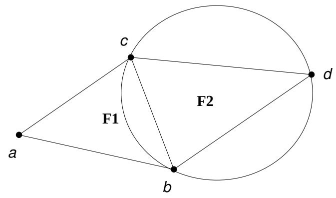
Figure 18. Operators used for inserting the edges of a closed trimming face.



(a) Adjacent faces.



(b) *Curvature* criterion.



(c) *Delaunay* criterion.

Figure 19. Geometric criteria for edge evaluation.

In Step 3c, the trimming regions are triangulated to complete the mesh reconstruction. An ear-cut algorithm[23] recursively subdivides each simple polygon representing a trimming face. The tests computed in parametric space to obtain an initial triangulation are:

Consistency. Edges should be contained completely in the parametric region limited by the edges, without intersecting other edges; and

Proximity. Edges should connect vertices that are near to each other.

The second stage of our triangulation algorithm is to perform edge-swap on the edges created in the first triangulation, until they obey the following geometric criteria:

Curvature. The sector formed by the chord and by the curve that follows the surface has least area.

Delaunay. Edges obey the standard Delaunay criterion projected on a plane whose normal vector is the average of the normal vectors of adjacent faces.

The *Curvature* criterion is evaluated with the aid of an adaptive polyline with the surface's curvature, generated to represent the edge in 3D. The area of the sector is approximated by the trapezoids formed by the projection of points on the chord (Figure 19b). The *Delaunay* criterion is evaluated on a plane whose normal vector is determined as an average of the normals evaluated at the middle points of both triangular adjacent elements, as shown in Figure 19a. The two adjacent faces are projected on this plane to check the angles at vertices a and d, shown in Figure 19c.

The triangulation algorithm uses these criteria in the order in which they were presented for the following reasons. First, the trimming regions may present sharp curvatures, where planar criteria (like *Delaunay*) may not give good results. In good finite-element meshes over surfaces with sharp curvatures, edges obey the *Curvature* more than the *Delaunay* criterion. Second, the *Curvature* criterion alone may not be adequate, as is the case of planar surfaces. Finally, trimming regions in planar surfaces subdivided by edges which only obey the *Consistency* criterion may be transformed into a constrained Delaunay triangulation by swapping edges[23] according to the *Delaunay* criterion.

For these reasons, the basic steps of the algorithm for triangulating a trimming face are:

1. Obtain an initial triangulation by using the *Consistency* and *Proximity* criteria by recursively subdividing the trimming face.
2. Swap the edges created in Step 1 according to the *Curvature* criterion. If the curvatures of both edges are identical, then use the *Delaunay* criterion.

Figure 20 shows the regions of Figure 16 triangulated by the algorithm using these criteria. The edges created to subdivide the trimming regions are drawn with thicker lines.

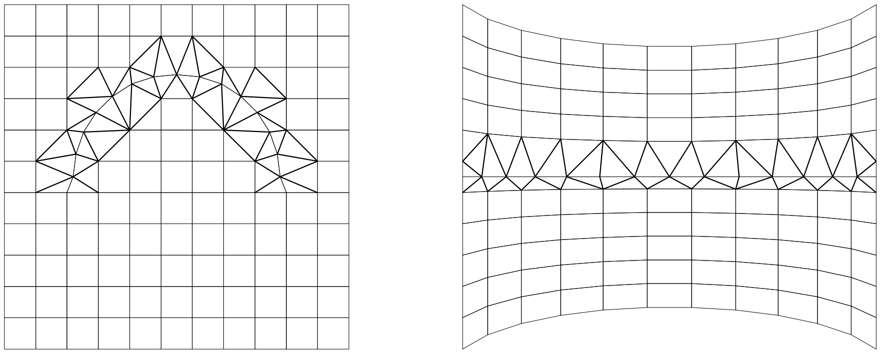


Figure 20. Reconstructed meshes after triangulating the trimming regions.

4.5. Smoothing

The reconstruction technique presented in previous sections produces good triangles in both meshes, as Figure 20 shows. Nevertheless, these meshes may be smoothed in order to improve the geometric quality of the elements. Here, it is again vital that we have a continuous parametric description for each surface, so that we can move vertices and still remain *on* the surfaces.

Rypl and Krysl[25] describe a methodology for smoothing meshes on surfaces using a pre-smoothing done in parametric space, followed by the 3D smoothing. The new positions defined by the usual Laplacian technique in 3D do not fall on the curved surfaces. Thus, a projection algorithm similar to the one used to relax the equally-spaced points over the trimming curves in Step 2d must be employed. The parametric pre-smoothing reduces the vertex/surface separations induced by the 3D Laplacian movements near regions where elements are not very stretched, thus speeding up the convergence of the projection algorithm.

In our implementation, we adopted three repetitions of the parametric smoothing and two repetitions of the 3D with projections. (Section 5.1 describes some test cases for the performance of this algorithm.) Figure 21 shows the resulting meshes after smoothing. Note that the vertices adjacent to a given vertex, as required by Laplacian smoothing, can be found easily with the DCEL data structure.

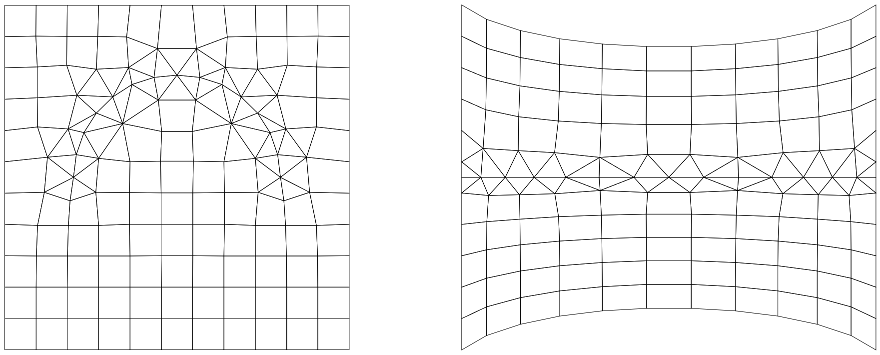
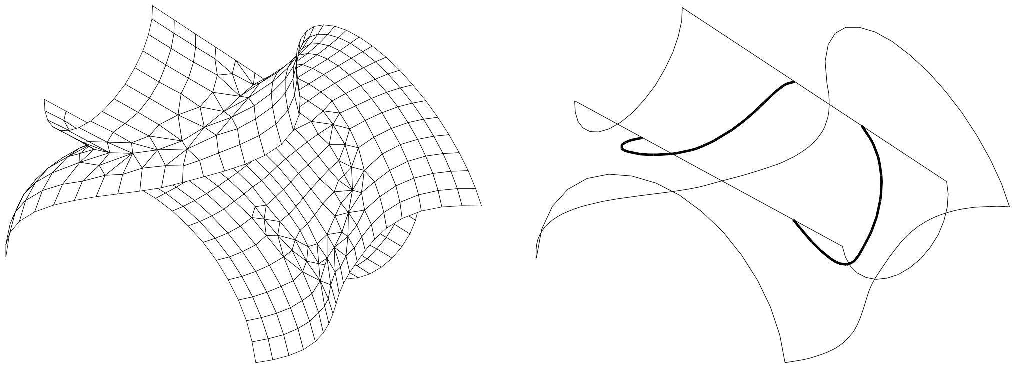


Figure 21. The meshes of Figure 20 after smoothing.



(a) Final mesh.

(b) Patch boundaries (thin lines) and intersection curves (thick lines).

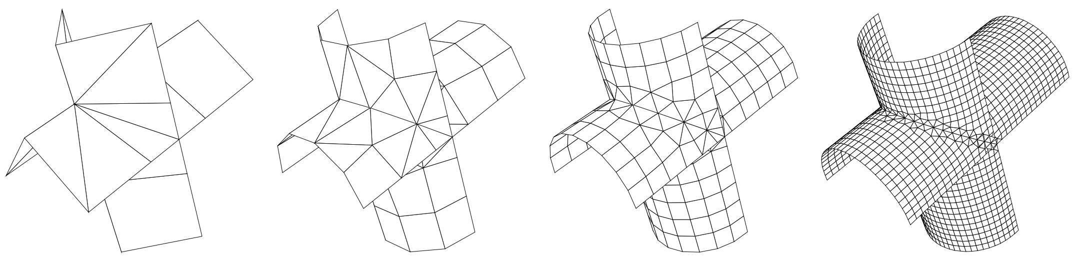
Figure 22. Intersection between cylindrical shell and bilinear Coons patch.

5. PAIRS OF SURFACES

This section shows the effectiveness of our algorithm in several examples of two intersecting surfaces.

The first example, which has already appeared in Figure 1, is revisited here and shown from a different viewpoint in Figure 22. The model is the union of a Coons patch with a cylindrical patch, whose bottom part has been trimmed away. Figure 22a shows a bottom view of the mesh, and Figure 22b shows the curves that model the shells (thin lines), and the two intersection curves (thicker lines).

The next example is the union of two cylindrical shells, modeled as bilinear Coons patches. Figure 23 shows the results of the algorithm for meshes of increasing complexity. In Figures 23a and 23b, the intersection region extends up to the boundaries, and smoothing repositions a few interior vertices. In higher-resolution meshes, only the few elements close to the intersection region are affected, and the original meshes are largely preserved (Figures 23c and 23d).



(a) 2×3.

(b) 5×5.

(c) 10×11.

(d) 30×30.

Figure 23. Intersections between cylindrical shells meshed with increasing resolution.

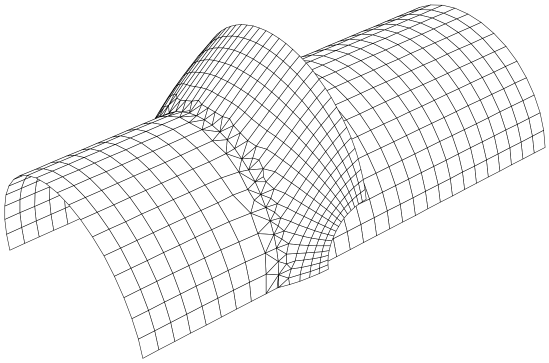
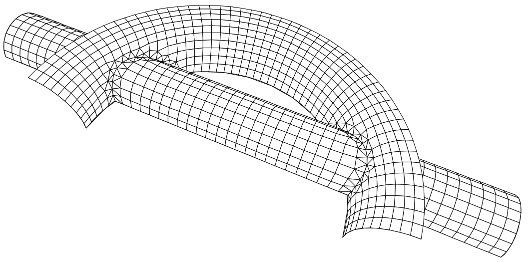


Figure 24. Intersection between a Coons patch and a cylindrical shell.

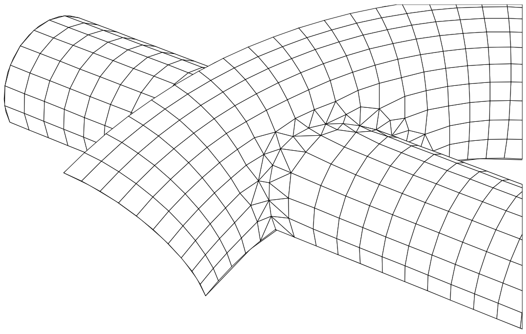
The effect of the local modifications can also be seen in a similar example, shown in Figure 24. This example is the basic intersection used in the modeling of the heat transfer shell described in Section 6.

Figure 25a shows an example of an intersection that has two connected components. The patches are a cylindrical shell and a toroidal shell. Figure 25b shows a closer detail. In this example, three distinct regions can be identified on each patch. Figures 26a and 26b show the elements in parametric space, with distinct regions drawn with different line widths. Note that the elements do not have good shapes in parametric space but they do in object space. Figure 26c shows the mesh resulting from removing the central region of the cylindrical shell.

The examples in Figure 27a and 27b show that the algorithm correctly handles closed intersection curves. The model shown in Figure 27a is composed by Coons patches where all boundary curves are arcs of circles. In Figure 27b, the central regions in both cylindrical shells were removed.

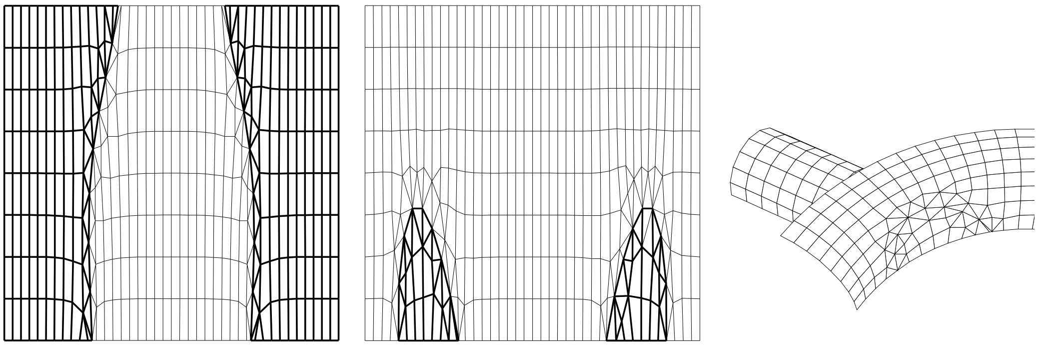


(a) The combined mesh.



(b) A closer detail.

Figure 25. The intersection between cylindrical shell and toroidal shell has two components.



(a) Cylindrical shell. (b) Toroidal shell. (c) Detail of the mesh after cut.

Figure 26. Distinct regions in parametric space and a detail of the mesh with the central part of the cylindrical shell removed.

5.1. Mesh quality analysis

To assess the quality of the meshes generated in the previous examples, we used the following criterion defined by Lau and Lo[8] for the geometric quality α of a triangle ABC:

$$\alpha(ABC) = 2\sqrt{3}\,\frac{\|AB \times AC\|}{\|AB\|^2 + \|BC\|^2 + \|CA\|^2}$$

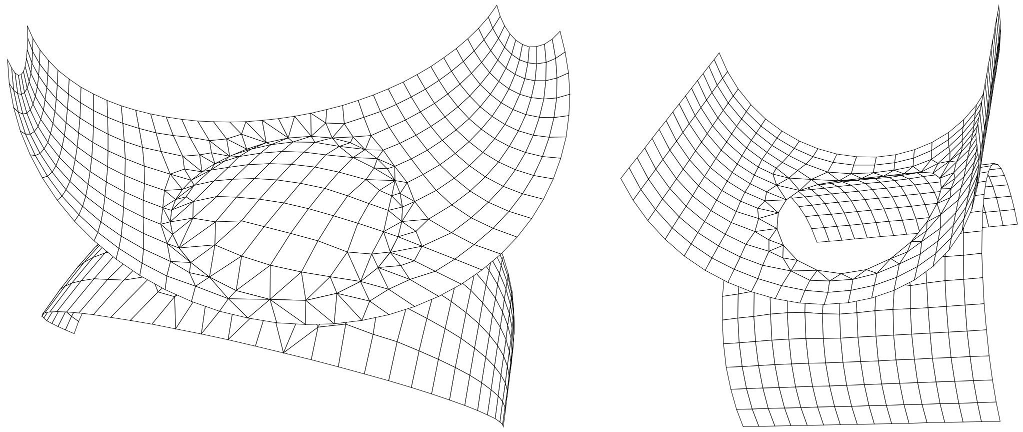
According to this criterion, the best triangles are equilateral triangles, which have $\alpha = 1$, and the worst triangles are degenerate triangles (whose vertices are collinear), which have $\alpha = 0$.

Table I shows the qualities of the examples shown in Section 5. From left to right, its columns are: references to the figures, the total number of elements in both meshes, the number of triangles created, the processing times in seconds of the whole algorithm obtained with a 200 MHz Pentium-Pro running Linux, and measures of how α is spread (average, maximum and minimum values, and standard deviations σ) before and after smoothing. Note how the meshes have improved after smoothing, in the sense that the average α has moved closer to 1, and the standard deviation σ is smaller.

To illustrate the quality enhancement of the triangular elements resulting from smoothing, Figure 28a shows the histogram of the α values for the triangles initially generated (before smoothing) over the cylindrical shells of Figure 23d. Figure 28b shows the histogram of the mesh of Figure 23d after the smoothing. These histograms were built with intervals of 0.05 α; the vertical axis shows the percentage of elements classified in each interval.

Figures 29a and 29b present the histograms of the meshes over the cylindrical shells shown in Figure 24. The histograms of Figure 29 use the same grouping described above for Figure 28.

Note the reduction of the dispersion (the standard deviations were greatly reduced), and the increase of the average element quality (the mean values were increased), although there are still some poorly shaped elements in the final meshes.



(a) Two Coons patches. (b) Two cylindrical shells.

Figure 27. Intersections that generate closed intersection curves.

6. COMPOSITE SURFACES

The first example of composite surface is a semi-submersible oil platform shell model used for stability analysis, shown in Figure 30. The model is a representation of the submerged faces for a given water plane and presents double symmetry, thus only a quarter of the complete shell was designed. The horizontal bracing intersects the vertical column, the surfaces over the pontoon, and the symmetry plane. The intersections were computed for each individual pair of surfaces; the vertices created over the boundaries were considered in subsequent intersections. The column intersected by the bracing is the one shown in Figure 2, as an example of bulging produced by reparametrization. Figures 30b and 30c show the details of this column with its meshes redefined by the intersections using the original parametrization.

A plane patch was modeled to represent the symmetry plane, and its intersections with the bracing were computed to cut the regions to the other side of this plane. Figure 31a shows the curves used to model the plane that cut the bracing. In Figure 31b, the curves used in the modeling of the shells are shown in thin lines, and the intersection curves are shown in thicker lines.

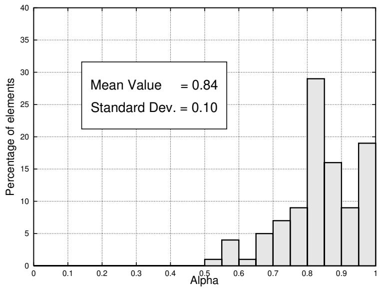
The second example of composite surface is a finite-element model of a heat exchanger shell built with the basic intersection shown in Figure 24. Figure 32a shows the final composite mesh. Figure 32b shows the curves used to model the individual patches (thin lines), and the intersection curves (thicker lines). The internal regions of the cylindrical shell and of the curved cylindrical shell were removed, after the computed intersections.

7. CONCLUSIONS

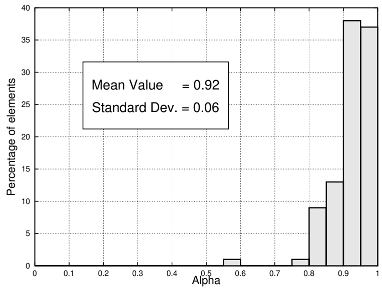
An algorithm that determines the intersection curves and the resulting meshes of two parametric patches was developed and implemented with the DCEL topological data structure, using spatial indexing trees for storing entities. These trees greatly reduce the complexity of intersection computations by quickly discarding large sections in one mesh that cannot intersect the other mesh. This enhanced

Table I. Number of elements, processing times, and mesh quality before and after smoothing.

Figure	elements	triangles	time (secs.)	α before smoothing				α after smoothing			
				ave	max	min	σ	ave	max	min	σ
1	931	260	25.36	0.81	1.00	0.20	0.15	0.90	1.00	0.55	0.09
23a	36	20	0.45	0.84	0.90	0.54	0.10	0.80	0.98	0.59	0.06
23b	50	52	0.68	0.84	1.00	0.54	0.10	0.90	1.00	0.73	0.06
23c	220	96	2.18	0.84	1.00	0.54	0.10	0.93	1.00	0.60	0.06
23d	1800	300	6.28	0.84	1.00	0.54	0.10	0.92	1.00	0.57	0.06
24	1550	274	6.19	0.80	1.00	0.18	0.17	0.90	1.00	0.21	0.10
25	728	170	4.48	0.77	1.00	0.01	0.23	0.91	1.00	0.10	0.18
27a	734	216	3.72	0.78	1.00	0.31	0.15	0.85	1.00	0.16	0.13
27b	952	284	6.45	0.76	1.00	0.16	0.20	0.79	1.00	0.43	0.12

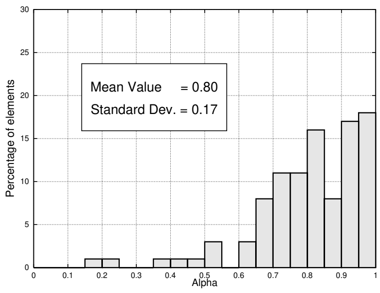


(a) Before smoothing.

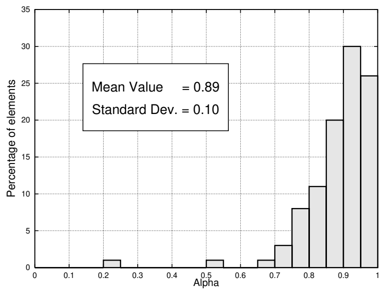


(b) After smoothing.

Figure 28. Histograms of the α values for the meshes of Figure 23d.

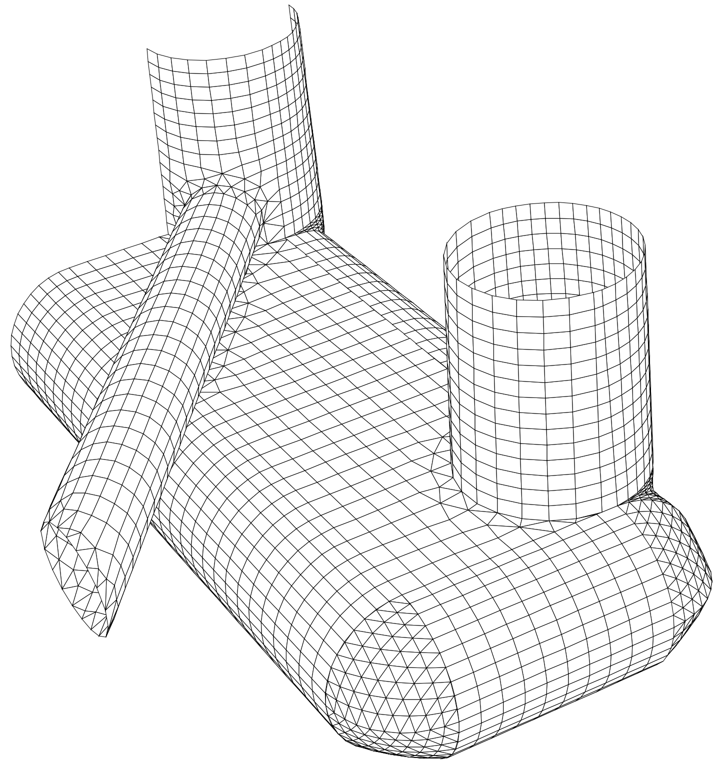


(a) Before smoothing.

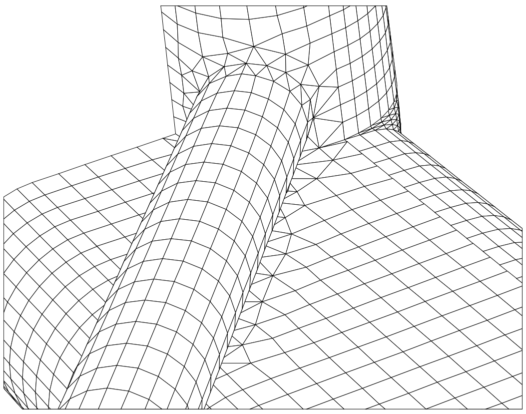


(b) After smoothing.

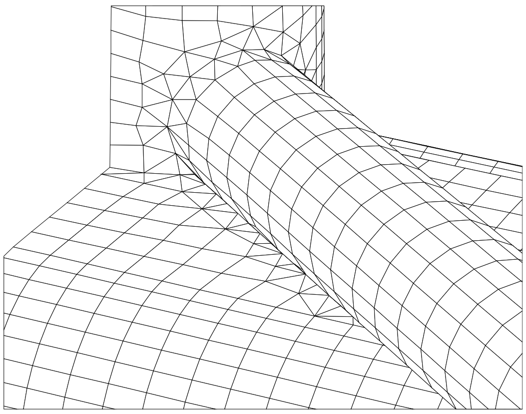
Figure 29. Histograms of the α values for the meshes of Figure 24.



(a) A complete view of the model.

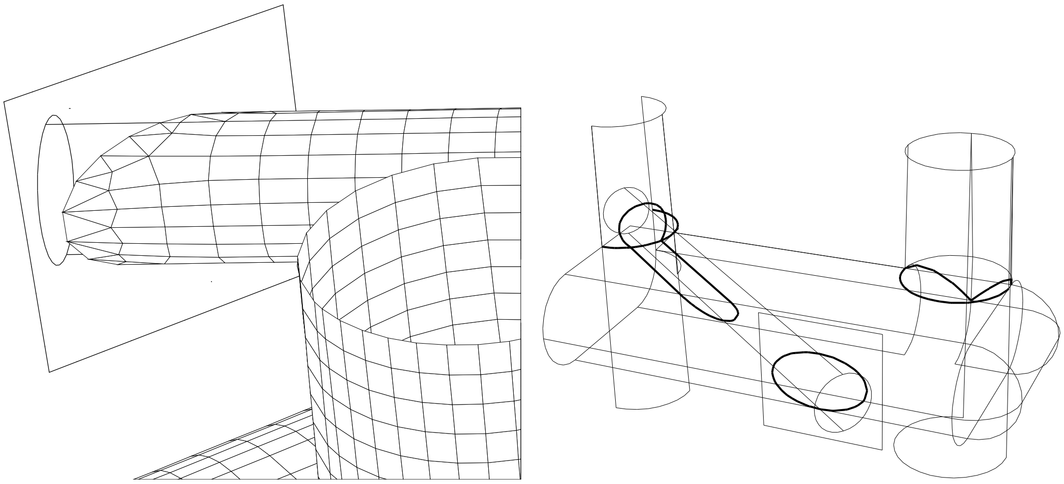


(b) A detail around the bracing.



(c) The other side of the detail.

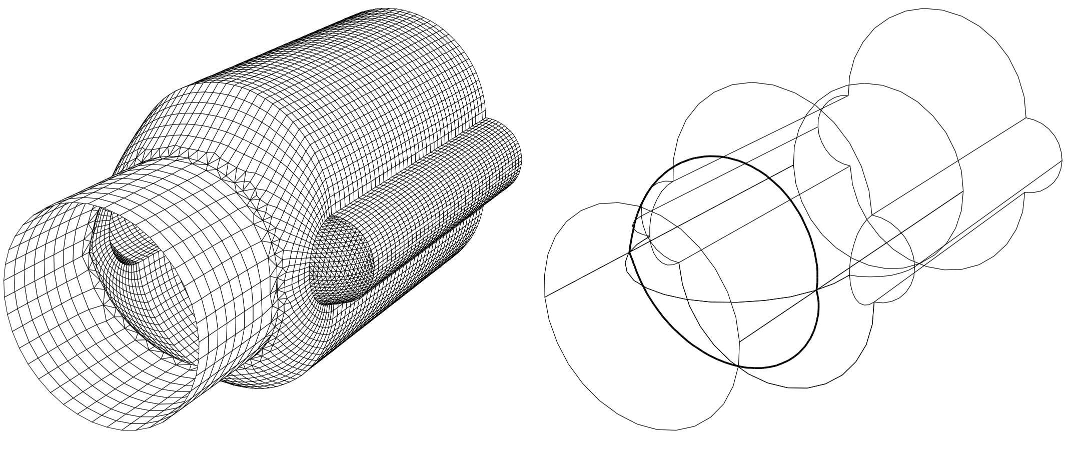
Figure 30. Semi-submersible oil platform shell.



(a) Curves of the symmetry plane patch.

(b) Curves used to model all patches.

Figure 31. Curves involved in the modeling of the semi-submersible oil platform.



(a) Final mesh of the heat exchanger.

(b) Curves used to model the heat exchanger.

Figure 32. A heat exchanger.

topological data structure allows the construction of the trimming curves without global searches. The topological information is also used in the re-meshing algorithm to obtain elements with improved geometric quality, suitable for finite-element analyses.

The DCEL data structure is simple and compact, and is a good choice for the shell modeling program. For applications that must deal with all topological degeneracies that appear in more general intersections (faces with holes, self-edges, or edges inside faces), the half-edge[26] data structure would be a better choice. However, the half-edge data structure is not compact and its impact on memory use would have to be further evaluated, specially for large meshes.

The results show that the time taken by the intersection algorithm is similar to the time required to model the individual patches, and thus can be used in interactive shell modeling with composite patches. Experiments with meshes having up to one million faces suggest that the time required by the algorithm grows linearly with the number of faces. This can be explained by the small number of face intersections actually processed due to the efficient searches provided by the spatial trees.

Step 3 of the algorithm is the most time-consuming of all three steps (taking an average of 60% of the total time), and its current implementation uses the speed-up described by O'Rourke,[23] which still is $O(n^2)$, where n is the total number of edges in the intersection region. In practice, n is much smaller than the total number N of edges in both meshes. Typically, $n = O(\sqrt{N})$, which is consistent with the experimental results mentioned above.

The algorithm does not need to order the intersection points in a face using geometric tests as done by Cavalcanti et al.[18] Unlike Lo's algorithm,[16] our algorithm does not require the global search for chaining the individual segments, because it performs this task as the global polylines are computed with topological tests on the extremities of the trimming curve (see Figure 14). It also handles trimming curves with sharp curvatures and several connected components.

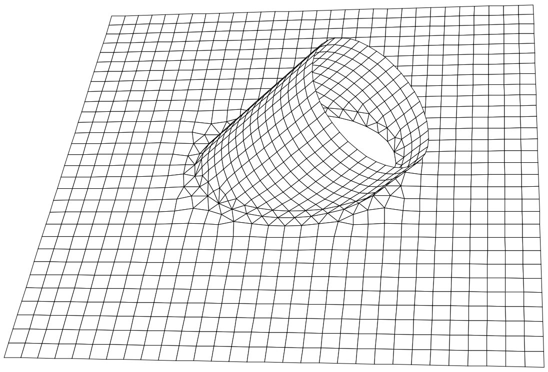
The geometric quality of the trimming curves generated by the algorithm was tested by comparing them with the analytical solution in some simple cases, with very good results. Thus, the interpolation method using Foley's parametric variation[24] presented good results in our tests.

One of the central problems faced by marching methods is the determination of the topology of the intersection curves:[12] one can jump from one component to another, if the intersection curves are too close. With the algorithm presented here, the topology of the components is well defined if the meshes have discretizations that are compatible with the algorithm restrictions (see Figure 11).

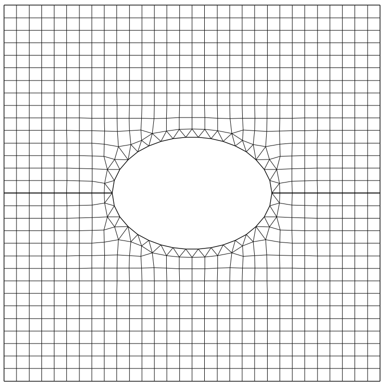
The sets of faces forming distinct regions, which are separated by trimming and boundary curves, can be identified easily despite the use of only one R*-tree per patch. Starting from any face from the region one wish to identify, the selection is propagated through the faces adjacent to the interior edges which have the value of trimming set to zero, that is, those which do not belong to the boundary nor to any trimming curve. This allows the specification of different attributes in an individual surface, and the trimming of excess parts.

The models shown in Figures 30 and 32 show that the algorithm can be used to model complex composite surfaces with few intersection operations. These examples also show that most elements of the mesh kept their original aspects, and those adjacent to the intersected elements had their vertices moved to enhance geometric quality.

We are currently working on mesh uniformization, converting triangular elements into quadrilaterals, using the method proposed by Potyondy et al.[27] The uniformization, however, should be done after an evaluation of the mesh quality in the sense of finite-element simulation, because geometric quality is only a necessary condition for good simulations. The adaptivity methodology presented by Riccius et al.[28] could be used to decide whether the improvements made to the meshes using the intersection algorithm presented here yield good results. Figure 33 shows the cylinder-plate intersection example evaluated by Riccius et al.[28] Figure 33a shows a combined mesh computed by our algorithm. Figure 33b shows the plane mesh isolated.



(a) Combined mesh of plane and cylinder.



(b) Mesh on plane.

Figure 33. Cylinder-plate intersection.

ACKNOWLEDGEMENTS

The MG shell modeling program has been developed for PETROBRAS in TeCGraf (the Computer Graphics Technology Group of PUC-Rio). Maurício Riguette Mediano implemented the R*-tree library and gave many useful suggestions. Waldemar Celes implemented the B-tree library. Leila de Floriani and Paola Magillo of the University of Genoa kindly shared their implementation of the topological operators shown in Appendix A. The authors are partially supported by research grants from the Brazilian Council for Scientific and Technological Development (CNPq). This research is part of the first author's doctoral thesis at PUC-Rio.

REFERENCES

1. G. Farin, *Curves and Surfaces for Computer Aided Geometric Design*, Academic Press, 1990.
2. D. Comer, 'The ubiquitous B-tree', *ACM Computing Surveys*, **11**, 121–131 (1979).
3. 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*, May 1990, pp. 322–332.
4. L. C. G. Coelho. MG: Mesh Generator. `http://www.tecgraf.puc-rio.br/~lula/mg`.
5. L. C. G. Coelho and C. S. de Souza, 'Comunicação de problemas e soluções geométricas em uma interface 3D', *Anais do VIII SIBGRAPI*, 1995, pp. 233–240.
6. R. Haber and J. F. Abel, 'Discrete transfinite mappings for the description and meshing of three-dimensional surfaces using interactive computer graphics', *International Journal for Numerical Methods in Engineering*, **18**, 41–66 (1982).
7. L. T. Souza and M. Gattass, 'A new scheme for mesh generation and mesh refinement using graph theory', *Computers & Structures*, **46**, 1073–1084 (1993).
8. T. S. Lau and S. H. Lo, 'Finite element mesh generation over analytical curved surfaces', *Computers & Structures*, **59**, 301–309 (1996).
9. R. E. Barnhill, G. Farin, M. Jordan, and B. R. Piper, 'Surface/surface intersection', *Computer Aided Geometric Design*, **4**, 3–16 (1987).
10. R. Barnhill and S. Kersey, 'A marching method for parametric surfaces/surface intersection', *Computer Aided Geometric Design*, **7**, 257–280 (1990).

11. Tz. E. Stoyanov, 'Marching along surface/surface intersection curves with an adaptive step length', *Computer Aided Geometric Design*, **9**, 485–489 (1992).
12. P. Schramm, 'Intersection problems of parametric surfaces in CAGD', *Computing*, **53**, 355–364 (1994).
13. E. Houghton, E. Emnett, R. Factor, and L. Sabharwal, 'Implementation of a divide-and-conquer-method for the intersection of parametric surfaces', *Computer Aided Geometric Design*, **2**, 173–184 (1985).
14. M. Gleicher and M. Kass, 'An interval refinement technique for surface intersection', *Proceedings of Graphics Interface '92*, 1992, pp. 242–248.
15. L. H. de Figueiredo, 'Surface intersection using affine arithmetic', *Proceedings of Graphics Interface '96*, 1996, pp. 161–170.
16. S. H. Lo, 'Automatic mesh generation over intersecting surfaces', *International Journal for Numerical Methods in Engineering*, **38**, 943–954 (1995).
17. J. Hoschek and D. Lasser, *Fundamentals of Computer Aided Geometric Design*, A K Peters, 1993.
18. P. R. Cavalcanti, P. C. P. Carvalho, and L. F. Martha, 'Nonmanifold modeling: an approach based on spatial subdivision', *Computer-Aided Design*, **29**, 209–220 (1997).
19. F. P. Preparata and M. I. Shamos, *Computational Geometry – An Introduction*, Springer Verlag, New York, 1990.
20. M. R. Mediano, M. Gattass, and M. A. Casanova, 'HPS-tree: Um método de acesso para armazenar mapas longos com multi-resolução geométrica e topológica', *Anais do IX SIBGRAPI*, 1996, pp. 219–226. `http://www.visgraf.impa.br/sibgrapi96/trabs/abst/a32.html`.
21. P. Deuflard, 'A modified Newton method for the solution of ill-conditioned systems of non-linear equations with applications to multiple shooting', *Numer. Math.*, **22**, 289–315 (1974).
22. J. J. Chen and T. M. Ozsoy, 'Predictor–corrector type of intersection algorithm for C^2 parametric surfaces', *Computer-Aided Design*, **20**, 347–352 (1988).
23. J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, New York, 1993.
24. T. A. Foley and G. M. Nielson, 'Knot selection for parametric spline interpolation', in L. Schumaker (ed.), *Mathematical Methods in CAGD*, Academic Press, 1989, pp. 445–467.
25. D. Rypl and P. Krysl, 'Triangulation of 3D surfaces', *Engineering with Computers*, **13**, 87–98 (1997).
26. M. Mäntylä, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, Maryland, 1988.
27. D. O. Potyondy, P. A. Wawrzynek, and A. R. Ingraffea, 'An algorithm to generate quadrilateral or triangular element surface meshes in arbitrary domains with applications to crack propagation', *International Journal for Numerical Methods in Engineering*, **38**, 2677–2701 (1995).
28. J. Riccius, K. Schweizer, and M. Baumann, 'Combination of adaptativity and mesh smoothing for the finite element analysis of shells with intersections', *International Journal for Numerical Methods in Engineering*, **40**, 2459–2474 (1997).
29. C. M. Hoffmann, *Geometric and Solid Modeling – An Introduction*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1989.

APPENDIX A. TOPOLOGICAL OPERATORS

The creation and maintenance of the parametric subdivision stored by DCEL is consistently performed by a set of *Euler operators* which maintain the *Euler-Poincaré* formula[26,29] ($V - E + F = 2$). This group of constructive and destructive operators covers all updates necessary to DCEL's structure inside the construction and intersection algorithms, as follows:

MSVF Operator Make Surface Vertex and Infinity Face initializes the construction by building the Surface entity, generating a vertex and the infinity face. The trees are also initialized and the infinity face is not inserted into the R-tree, as it is a special entity with an orientation opposite to the others. Euler's number is kept as: $1 - V + 0 = 1$.

MEV Operator Make Edge and Vertex creates a new vertex and a new edge that connects the vertex to the rest of the structure.

MEBF Operator Make Edge Bound Face creates the first face of the representation by dividing the infinity face into two, inserting a new edge. This special operator is activated only once during the whole construction, and it is presented separately because it has totally different implementation and functionality from the traditional MEF.

MEF Operator Make Edge and Face creates an edge which connects two existing vertices, dividing a face which is passed as a parameter divided in two. It is used to create the faces following the face created by MEBF.

KEF Operator Kill Edge and Face (opposite to MEF) is used to remove an edge and, consequently, a face. The edges that used to point to the removed face will point to the one beside the removed edge.

KEV Operator Kill Edge and Vertex (opposite to MEV) eliminates an edge and a vertex, which is simply connected to the rest of DCEL by this edge.

KSVF Operator Kill Surface Vertex and Infinity Face is the contrary to MSVF, being used for the final removal of the structure.

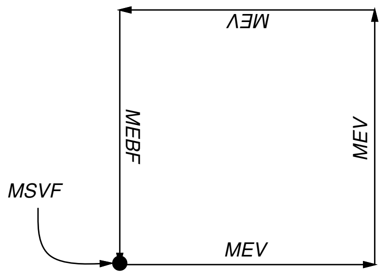
SEMV Operator Split Edge and Make Vertex is used to divide edges belonging to the boundary or to a trimming curve inside a patch, in Step 3b of the algorithm, as presented in Section 4.4.

APPENDIX B. INITIAL SURFACE MAPPING

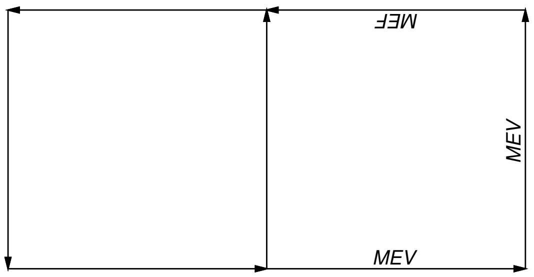
The construction of the initial surface mappings can be done starting from the classical finite-element structure: list of vertices and faces. Each face of the list contains one cycle of vertices, and the vertices contain the geometric coordinates and an index which identifies them in the face cycle. The vertex coordinates, however, must be identified in the parametric space of the surfaces, and not in 3D, as it happens in models for finite elements.

The determination of the correct Euler operator to be used and of the entities which must be updated are performed with searches in the vertex B-tree, outside the Euler operator's algorithms. The sequence is defined for each element's edge by testing the existence of its two vertices.

Figure 34a shows the operators needed for the creation of the first face of the representation. Figure 34b shows the operators needed for the insertion of the second and subsequent faces.



(a) Initial Face.



(b) Second Face.

Figure 34. Operators used for initial face assembly.