INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
Int. J. Numer. Meth. Engng (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/nme.4385

Repairing unstructured triangular mesh intersections

David McLaurin[1,*,†], David Marcum[2], Mike Remotigue[1] and Eric Blades[3]

[1]*Center for Advanced Vehicular Systems, Mississippi State University, MS, USA*
[2]*Department of Mechanical Engineering, Center for Advanced Vehicular Systems, Mississippi State University, MS, USA*
[3]*ATA Engineering Inc, Huntsville, AL, USA*

SUMMARY

Computational analysis and design has become a fundamental part of product research, development, and manufacturing in aerospace, automotive, and other industries. In general, the success of the specific application depends heavily on the accuracy and consistency of the computational model used. The aim of this work is to reduce the time needed to prepare geometry for volume grid generation. This will be accomplished by developing tools that semi-automatically repair discrete data. Providing another level of automation to the process of repairing large, complex problems in discrete data will significantly accelerate the grid generation process. The developed algorithms are meant to offer a semi-automated solution to a complicated geometrical problem — specifically discrete mesh intersection.

The intersection-repair strategy presented here focuses on repairing the intersection in-place as opposed to rediscretizing the intersecting geometries. Combining robust, efficient methods of detecting intersections and then repairing intersecting geometries in-place produces a significant improvement over techniques used in current literature. The result of this intersection process is a nonintersecting geometry that is free of duplicate and degenerate geometry. Results are presented showing the accuracy and consistency of the intersection repair tool. Copyright © 2012 John Wiley & Sons, Ltd.

Received 14 January 2011; Revised 12 April 2012; Accepted 3 June 2012

KEY WORDS: unstructured; mesh; intersection; repair; healing

1. INTRODUCTION

Computational analysis and design has become a fundamental part of product research, development, and manufacturing in the aerospace, automotive, and other industries. The process typically begins with the construction of a computational model to use as a virtual representation of real-world geometry. For many downstream applications, such as computational fluid dynamics, computer graphics, structural analysis, or simulation of manufacturing processes, this is often performed with a computer-aided design (CAD) system. Thus, the computational model is commonly called a CAD model. This computational model is a starting point from which simulation or analysis can be performed. Each application has a field-specific set of requirements based on the physics and numerical processes involved. In general, the success of the specific application depends heavily on the accuracy and consistency of the computational model used. With the ever-increasing power of modern computers and the relatively automatic nature of mesh generation and computational field simulation, the less automated, user-intensive CAD model generation and model repair has begun to dominate the amount of time required for performing a computational simulation. By some accounts, it is estimated that 75% of the man-time for an overall numerical simulation is spent during the geometry preparation, repair, clean-up, and mesh generation [1].

*Correspondence to: David McLaurin, Center for Advanced Vehicular Systems, Mississippi State University, MS, USA.
†E-mail: dom9@cavs.msstate.edu

The choice between CAD-based repair techniques and discrete-geometry-based repair techniques was made by recognizing the context in which developed techniques would be used. Repairing CAD-based geometry can generate many numerical inaccuracies because of nonlinear operations, such as intersections and projections using high-order nonuniform rational b-splines. However, performing intersections or projections using discrete geometry involves linear operations using linear elements — at least for this application. A discrete representation can also be used as an underlying geometric representation for downstream applications such as mesh generation. The purpose of this research is to accelerate the process of generating a watertight, manifold grid no matter its origin. This will be accomplished by accelerating the discrete-geometry repair and clean-up processes through discrete-geometry-based repair techniques.

In this work, algorithms were developed to repair CAD models that exhibited specific problems that are unable to be fully automated — specifically discrete mesh intersections. Techniques that remesh the problem areas via volumetric techniques or consistent boundary application will not be considered here. These methods offer a large amount of automation at the potential expense of mesh accuracy and repair time. Instead, methods of repair have been developed that directly alter existing topology to repair intersections. The developed algorithms are meant to offer a semi-automated solution to a complicated geometrical problem. These techniques were developed to be controlled by a user because the problems that are addressed are not minor, but major imperfections that cannot be repaired automatically.

1.1. Proposed algorithm for repairing discrete mesh intersections

Because collision detection is so widely used in areas such as video games, efficient and accurate methods of detecting intersections in various types of meshes have been developed. However, once the intersections are detected, current methods of repairing the intersecting geometry focus mainly on rediscretizing the areas found to intersect [2]. In addition, current methods are usually restricted to geometries that are already watertight [3].

The work presented here focuses on repairing the intersection in-place as opposed to rediscretizing the intersecting geometries. Repairing the mesh in-place entails directly altering the geometry topology. This removes the potentially large expense associated with the rediscretization process and lifts the requirement of the input mesh being watertight. Intersections are detected through the use of an octree data structure [4], in which the discrete elements are stored. This significantly reduces the amount of discrete element–element tests required to detect intersections. Once detected, the intersections are repaired by calculating lines of intersection between intersecting discrete elements and inserting them into the geometry by using element or edge splitting. The topology present in the model is then used to find intersections of elements that are topologically adjacent to the originally detected intersection — further reducing the amount of discrete element–element intersection tests.

Accurate and reliable calculation of the lines of intersection is also vital to the success of the tool. The intersection tests and subsequent edge insertions are performed using localized tolerances and topological primitives. This not only increases the robustness of the algorithm, but also does not require user intervention. Combining the robust, efficient methods of detecting intersections and then repairing intersecting geometries in-place produces a significant improvement over techniques used in current literature [2,3,5]. The result of this intersection process is a nonintersecting geometry that is free of duplicate and degenerate geometry. This paper is organized as follows: discussion of triangle–triangle intersection tests, application of neighbor tracing to this problem, and local repair process. Some results demonstrating the various aspects of the tool will also be presented.

2. MESH INTERSECTION

2.1. Triangle intersection tests

Two triangle–triangle intersection (TTI) tests were found to be the most widely used in relevant literature, [3,6], and therefore will be discussed here. For the purposes of describing TTI tests, let us denote the two triangles T_0 and T_1, respectively. Also, let us state that for two triangles to intersect in three dimensions, the following two conditions must exist: two edges of each triangle must cross the plane of the other, and if so, then two edges must intersect the aforementioned planes within the boundaries of the triangles. This definition is important later for determining degeneracies.

One type of TTI, first developed by Moller [6]and used by Lo and Wang [2], tests for intersections using planes and signed distances. This intersection test takes advantage of the fact that the intersection of two planes is a line [7]. Triangles are also planar objects, so this test computes the plane of T_0, denoted P_0, and the plane of T_1, denoted P_1. The line segments that define the intersection of T_0 and P_1 and the intersection of T_1 and P_0 are first calculated. If the two line segments overlap, then the triangles intersect. Problems that exist with this method include the need to trap out zeros — which can be caused by large differences in scale, nearly degenerate geometry, or nearly coplanar geometry. The floating-point division required by this approach may result in overflow and subsequently cause serious problems with robustness.

The other type of TTI, demonstrated by Aftosmis [3], involves a Boolean check for intersection that only involves multiplication and division and does not involve expensive operations like square roots or trigonometric functions. Once the triangles are found to be intersecting, the end points of the line segment defining the intersection can be calculated. This approach lessens or eliminates the problems with the aforementioned method because the intersection line-segments are only calculated for geometry that is known to intersect. The aforementioned Boolean test involves the calculation of the signed volume of a tetrahedron. This Boolean test constitutes a topological primitive[8] and is the fundamental building block for all TTI tests performed in this research.

A topological primitive is defined in [3] as, ‘an operation that tests an input and results in one of a constant number of cases’. It is further stated that, ‘Such primitives can only classify, and constructed objects (like the actual locations of the points of intersection…) cannot be determined without further processing. These primitives do, however, provide the intersections implicitly, and this information suffices…’. In this case, the constant number of cases that can be returned from the volume calculation is three: positive (+), negative (–), or zero. Positive and negative results represent nondegenerate cases and zero represents some degeneracy involved with the geometry, for example, edge–edge intersection or point–edge intersection. By defining ‘zero’ locally for each pair of triangles tested for intersection, this tool becomes very robust. Because the information pertaining to whether geometry is degenerate with respect to an intersection is used, the need for computationally expensive, exact-arithmetic routines is avoided. See the section on robustness for a more detailed explanation on how computational errors associated with degenerate geometries are handled.

2.2. Neighbor tracing

Lo and Wang [2] presented a method for further reducing the cost of repairing intersecting triangular meshes. The intersection between discrete surfaces is defined by a set of connected line segments. Each pair of triangles that intersect contributes one line segment to this set. Instead of relying solely on a spatial subdivision scheme to reduce the number of TTI tests performed, Lo and Wang [2] proposed that once a pair of intersecting triangles was found, the topology of the mesh be used to construct the set of line segments defining the intersection. They denoted this process as ‘tracing neighbors of intersecting triangles’ (TNOIT). TNOIT involves first finding a pair of intersecting triangles, and then the topological relations in the mesh can be used to move along the lines of intersection in the mesh; — further reducing the number of TTI tests required to repair the mesh. Lo and Wang [2] presented three fundamental types of triangle intersections. In addition to those three intersection types, others, which include degenerate geometries, were developed here. As can be seen in Figure 1, an edge might not pierce within the boundaries of a triangle. If it does not, then it either it must pierce an edge of the triangle, or an edge pierces a node of the triangle. Each of these requires different methods of moving to the next pair of intersecting triangles. In Figure 1(a), an edge of T1 intersects an edge of F1. This means that both edges would have to be traversed to move to the next pair of intersecting triangles. In Figure 1(b), an edge of T1 intersects a node of F1. This means the element topologically adjacent to T1 would have to be tested against every element attached to the intersecting node, P, to move to the next pair of intersecting triangles.

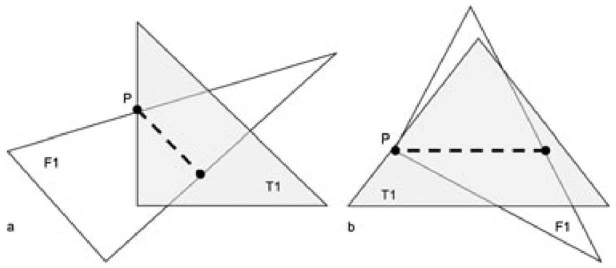

Figure 1. Degenerate possibilities of intersections.

2.3. *Local repair*

While tracing the intersection through the mesh, the lines of intersection that lie in each triangle are stored for later use. These line segments represent the intersection between the two surfaces. To repair the intersection, the line segments must be inserted into both surfaces. The process of inserting these line segments into the surfaces is simplified by the realization that each triangle has a set of edges that need to be inserted locally. This realization and subsequent simplification of the complex issue of repairing intersecting discrete geometries is one of the chief contributions of this work. This means that instead of a global set of edges to insert into the mesh, the problem can be broken into many sets of edges inserted into one triangle locally. Inserting edges in a triangle is strictly a two-dimensional task and no attempt to make a three-dimensional generalization of this procedure is made here.

A temporary, two-dimensional mesh is constructed out of the triangle and the nodes that define the edges by what is effectively a coordinate transformation. This local, two-dimensional transformation is accomplished by an affine rotation of the geometry into a local x–y plane defined at the centroid of the triangle. Given an origin and an axis of rotation, the transformation matrix is well known and is of the form found in [9].

By rotating the geometry, instead of projecting it, or using any other means, the undistorted geometry is transformed into two-dimensional space. This is important because if the wrong geometry were created in two-dimensional space as a result of an incorrect transformation, the resulting three-dimensional geometry would also be incorrect. An example of the rotated geometry, including the triangle and the to-be-inserted edges can be seen in Figure 2(a). The edges are inserted individually by first inserting the defining node(s) and then recovering the edge, Figure 2(b). The defining nodes can be inserted in a triangle or on an edge. If a containing triangle is found, it is split into three

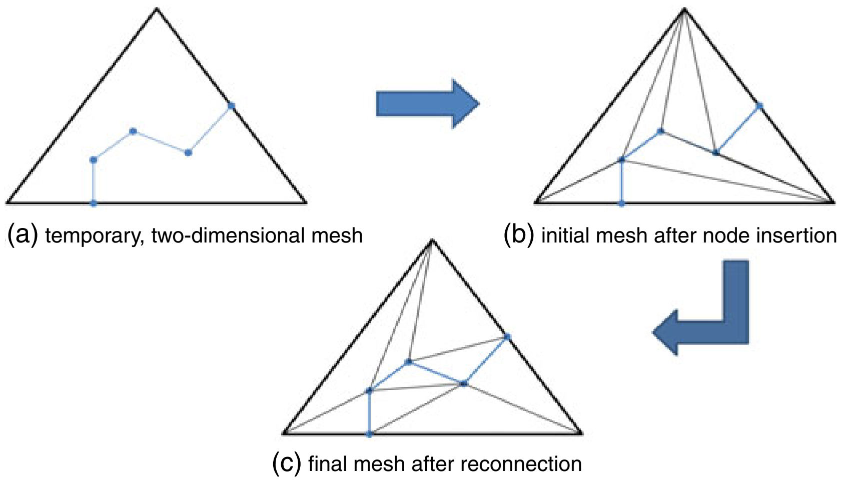
(a) temporary, two-dimensional mesh
(b) initial mesh after node insertion
(c) final mesh after reconnection

Figure 2. Local repair view edge insertion.

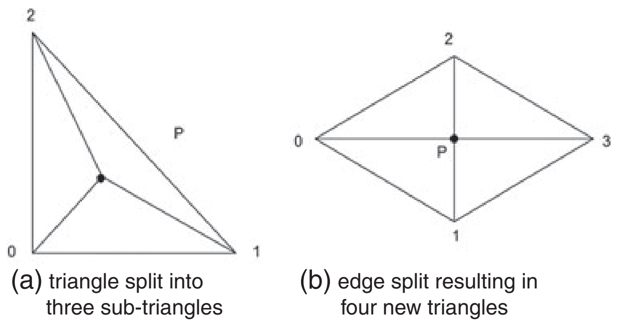
2
P
0
1
(a) triangle split into three sub-triangles
2
0
P
3
1
(b) edge split resulting in four new triangles

Figure 3. Triangle splitting and edge splitting example.

triangles, as seen in Figure 3(a). If the node is found to lie on an edge, the edge is split as seen in Figure 3(b). The original geometry was comprised of triangles (0,1,2) and (1,2,3). The node was found to lie on edge (1–2) and the edge was subsequently split to arrive at the geometry shown below. Four triangles now exist, (0,1,P), (0,P,2), (1,P,3), and (P,3,2). It is possible to not include the edge splitting option and rely on a local reconnection routine to improve mesh quality. However, the creation of nearly degenerate geometry by inserting nodes that are close to edges might cause the subsequent node insertions or containing-triangle searches to fail. Nearly degenerate geometry could also cause incorrect results from numerical inaccuracies. Therefore, the edge–split option was included.

2.4. Edge recovery and mesh quality

Once the defining nodes of an edge are successfully inserted into the two-dimensional triangulation, the edge itself must be recovered. It has been proven that the recovery of an edge in two dimensions is always guaranteed through a topological operation called edge swapping [7]. Edge recovery is a well-understood problem — especially in two-dimensions — and an example of this kind of edge recovering algorithm can be found in [10]. Once all of the required edges have been recovered in the temporary mesh, a constrained min–max (minimize the maximum angle) reconnection algorithm is used to improve the element quality in the local mesh. The aforementioned constraints are the inserted edges. These edges must be present for the final geometry to repair the intersection.

2.5. Translating local to global

As stated previously, a two-dimensional mesh was created for the purposes of simplifying the process of inserting nodes and subsequent edges into individual triangles. After all of the nodes have been inserted and edges recovered in the two-dimensional mesh, the topology of the two-dimensional mesh is used to replace the topology of the three-dimensional mesh without any further transformations. The topology of the two-dimensional mesh will be identical to the topology of the desired three-dimensional geometry. Therefore, no additional calculations are needed to transform the temporary mesh back to three dimensions — only the connectivity from the two-dimensional mesh.

2.6. Post processing intersecting mesh

The process of inserting the line segments, or edges, defining the intersection into all of the appropriate discrete surfaces necessarily creates nonmanifold edges where the intersections occur in the mesh. The purpose of this tool is to aid in the production of watertight, manifold meshes. Therefore, some way of removing the undesired geometry needed to be developed; otherwise, the intersection has removed one problem, intersecting geometry, and created another, nonmanifold edges. In general, it would be impossible to develop criteria that would remove unwanted geometry after intersection. Therefore, this process is left up to the user and facilitated through a surface painting algorithm that is bounded by nonmanifold edges. Surface boundaries are defined here as the edge-boundary between sets of triangle patches that are grouped, for example, wing or fuselage. Free boundaries are defined here as edges with only one topologically adjacent triangle. This post-processing step of surface painting would, if possible, separate the surface patches so that they are defined by surface patch boundary edges, free boundary edges, and the nonmanifold edges previously created by the mesh intersection routines. These separated surfaces can be removed or kept by the user based on the desired results. The results from this algorithm are demonstrated visually in the results section.

2.7. Robustness — triangle intersection test

In an attempt to increase the robustness of this tool, tolerances for determining degenerate geometry are determined locally. This method not only allows the mesh to be of any scale or order that is representable, but also prevents the user from having to enter a tolerance or even know anything about the scale or order of the mesh. For each TTI test, a number of volumes must be calculated. Instead of comparing the volumes to machine-zero for the purposes of determining geometric degeneracies, they are compared with an idealized volume that is calculated for each pair of triangles. In this implementation, this idealized volume is a bounding box, a cube in this case, formed by the longest edge of the six that are present in the triangle pair. A volume calculation is determined to be degenerate if it falls below a set fraction of the idealized volume. The purpose of this is that, in this case, a global tolerance is meaningless unless it is given some scale. This is because the use of a global tolerance could lead to very poor quality triangles being created because geometry may or may not be locally degenerate based on a global tolerance. A local tolerance is used to capture degenerate cases of intersection (see Figure 1). An example of a degenerate case is an edge–edge intersection. Instead of creating geometries with edges that have insignificant scale when compared with the surrounding geometry, the edges are split to arrive at a better mesh quality around the intersection.

A degenerate volume will be treated differently depending on when it is encountered. For example, the TTI test consists of essentially two steps: for two triangles, T_0 and T_1, test edges of T_0 to see if they pierce the plane of T_1 and, test edges of T_0 to see if they pierce within the boundaries of T_1. If a degeneracy is encountered while testing for edges piercing a plane, it indicates that the edge might not pierce the plane and subsequently the TTI test would fail. A technique called 'simulation of simplicity' [3, 8] is used to 'break' the degeneracy, or tie. This technique virtually perturbs the topological primitive with a unique perturbation dependent on node index and coordinate dimension. No changes are made to the actual geometry because the perturbation is applied to the result of the volume calculation. The virtual perturbation consists of components of decreasing magnitude that are considered one by one to 'break' the degeneracy or tie. If a degeneracy is encountered when testing if the edges of T_0 pierce within the boundaries of T_1, the edges might intersect within some tolerance and constitute a local degeneracy. This information is used to continue the TNOIT process.

Because the tolerance used for determining intersecting edges is not zero, the edges will most likely not intersect. Therefore, an equation for finding the point of intersection between two line segments is not applicable. The solution to this problem is to instead calculate the closest point to each edge on each edge. It does not matter on which edge the point resides because, if they intersect, they will both be split with the same point. The addition of edge splitting produces higher quality elements along the line segments of intersection.

3. RESULTS

In previous chapters, the algorithms for repairing mesh intersections and isolated boundaries are described in detail. This chapter demonstrates results from using these algorithms to repair three-dimensional geometry. All of the tests present in these results were run on a system with 2.93 GHz Intel Core i7 and 16 GB of RAM.

3.1. Bunny-Dragon

To demonstrate the accuracy and robustness of the developed algorithm, the following geometries were chosen because of their ubiquitous nature, complexity of intersection, and the large difference in relative mesh quality and mesh resolution. The Stanford Bunny and the Stanford Dragon [11] are shown individually in Figures 4(a) and (b), respectively.

These geometries were concatenated to form the geometry shown in Figure 5. The Stanford Bunny was used at the resolution of 35,947 points and 69,451 triangles. The Stanford Dragon was used at the resolution of 437,645 points and 871,414 triangles. These two models, while similar in absolute scale, have much different mesh resolutions. Because these models were created from scanning real objects, they have texture and roughness not present in contrived examples and are therefore prime candidates for demonstrating the aspects of the developed algorithms. The time required to perform all six intersections associated with this model is under 1 min.

As can be seen in Figure 5, the intersection of the Bunny and Dragon would result in each model being separated into several pieces. After the two were intersected the resulting mesh was separated using the aforementioned surface painting algorithm. The Bunny has been split into six pieces — five of which (green) were created because of the intersection with the Dragon. The green sections are on the interior of the Dragon and the yellow sections are the exterior (whetted) surface. These results can be seen in Figure 6. Figure 7 shows close-up views of some areas of

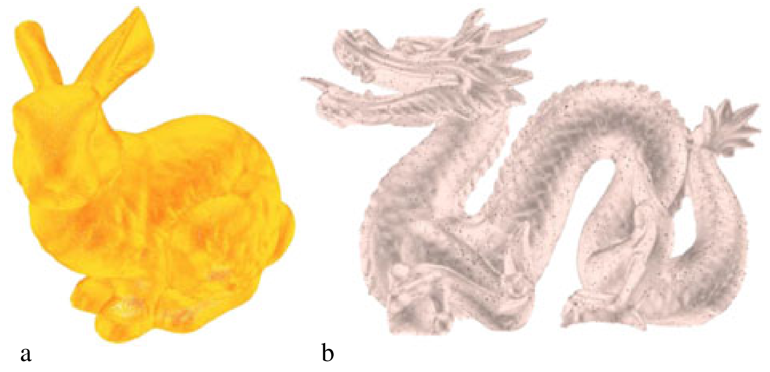

Figure 4. (a) Stanford Bunny and (b) Stanford Dragon.

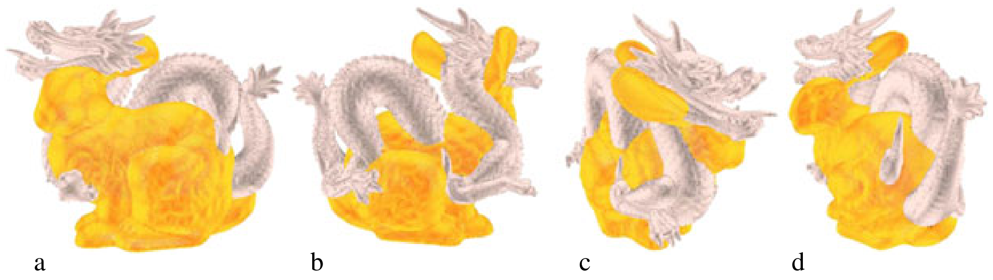

Figure 5. Concatenated Bunny-Dragon mesh: (a) front, (b) back, (c) head, and (d) rear.

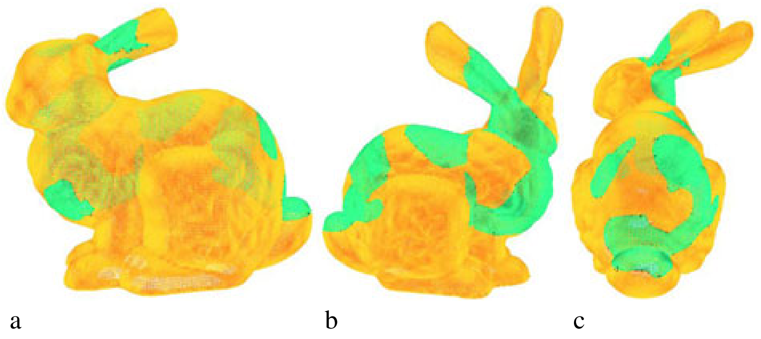

Figure 6. Bunny after intersection: (a) front, (b) back, and (c) rear.

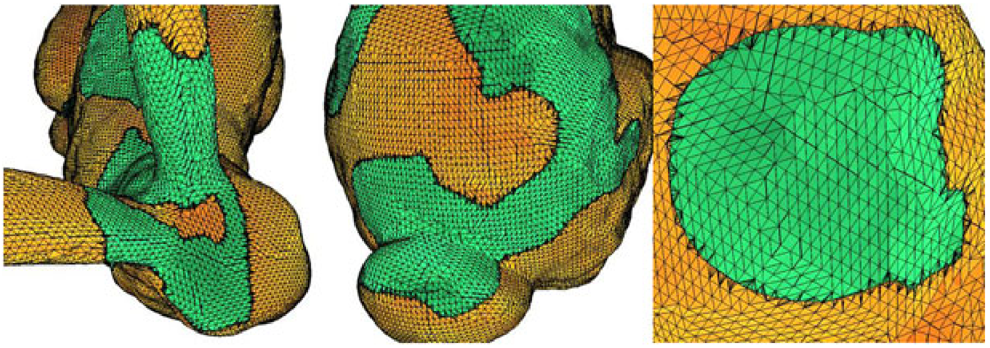

Figure 7. Bunny close-up after intersection: (a) top of head, (b) tail, and (c) front-left side.

interest — particularly the top of the Bunny's head (where it intersects with the Dragon's mouth), the Bunny's tail (where it intersects with the Dragon's tail and hind-right foot), and the Bunny's front-left side (where the Dragon's front-left foot protrudes). These were chosen to show the capture of the complex outline of the intersection (Figures 7(a) and (b)) and the resultant mesh region after intersection (Figure 7(c)). In Figure 7(c) the detail of the repaired region can be seen. The mesh that is topologically adjacent to the triangles that formed the intersection was untouched during the repair. Only the intersecting triangles were split to repair the intersection, that is, the mesh was repaired in-place instead of remeshing.

Figure 8 shows the Dragon after intersection. The resultant mesh has been split into six pieces — five of which were created because of the intersection. The green sections are on the interior of the Bunny and the silver sections are on the exterior (whetted) surface. Figure 9 shows close-up views of some areas of interest, particularly the underside of the Dragon's mouth (where

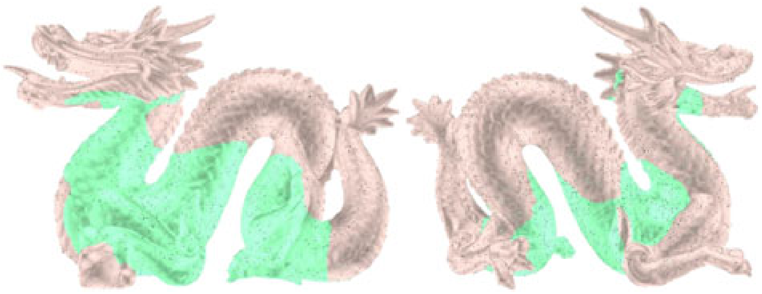

Figure 8. Dragon after intersection: (a) front and (b) back.

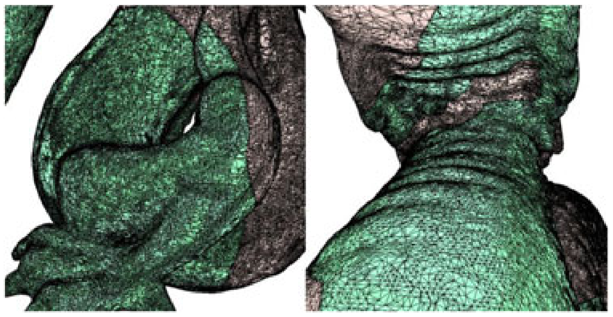

Figure 9. Dragon close-up after intersection: (a) rear-right foot and (b) under mouth.

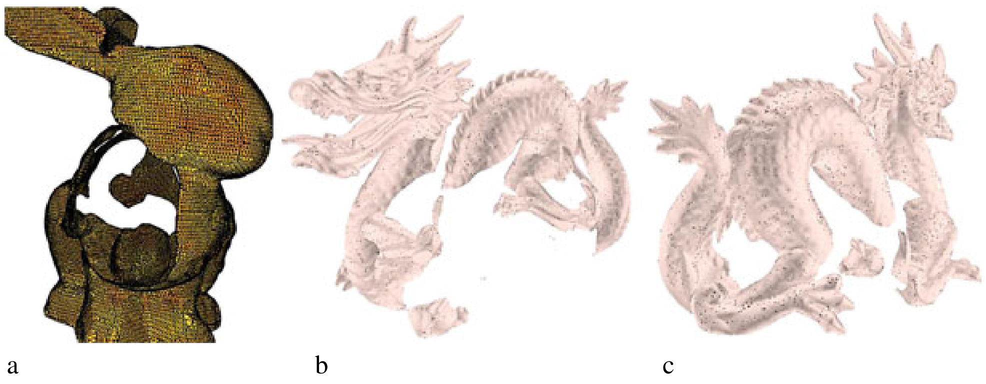

Figure 10. (a) Bunny and (b,c) Dragon with intersected pieces removed.

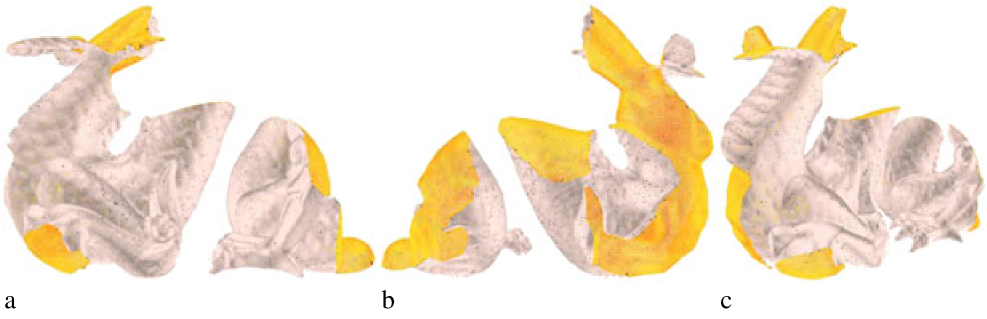

Figure 11. Interior of Bunny (yellow)-Dragon (silver) intersection: (a) front, (b) back, and (c) isotropic.

it intersects the Bunny's head) and the Dragon's rear-left foot (where it intersects with the rear-left portion of the Bunny). These were chosen to highlight how accurately the intersection was captured despite the relatively large difference in mesh resolution and mesh quality.

Figure 10 shows the exterior (whetted) surface of each model after intersection. The Bunny has large holes where the Dragon intersects the body and small sections of the ears and head. The Dragon also has large areas of the mesh missing where it intersects with the Bunny. The combination of these two models with the interior sections removed represents the whetted surface of the combined models after intersection. Once intersected, they are topologically adjacent and can therefore be treated as combined or remain as separated data. This is due to the fact that the lines of intersection were inserted into both meshes, not one or the other, to repair the intersection.

Figure 11 shows the mesh with the exterior (whetted) surfaces removed to reveal only what is interior to both the Bunny and Dragon. The exterior would appear as in Figure 5. If the intersection (in the set-theory sense, data that are only in the interior to both sets of data) of the two sets of data is desired, this can also be obtained by removing the exterior surface patches. Each of the models now has been separated into pieces that are bounded by the lines of intersection between the two models. These surface patches are distinctly marked so that they can be removed or kept based on what is desired. These surface patches have been shaded appropriately here for visual purposes. For instance, the Bunny in Figure 6 (after intersection) is now comprised of six distinct surface patches that are topologically adjacent along the lines of intersection. The Dragon has also been split into six distinct surface patches.

4. CONCLUSIONS

Methods for semi-automated repair of intersecting triangular meshes and isolated free-boundaries were designed, developed, implemented, and validated for three-dimensional meshes. Results show that these algorithms repair the models while maintaining small features and curvature present in the original data. The methods were shown to be robust by demonstrating correct results even when the meshes were of varying resolution and element size. It is evident from the results that these tools could substantially reduce the time and cost associated with manual mesh repair.

Algorithms to repair intersecting triangular meshes were developed. The intersection was found through the use of an octree. This particular spatial subdivision strategy offered the advantage of reducing the number of intersection candidates possible for each triangle. Once an intersection was found, topology information was used to calculate a set of connected line segments that forms the intersection between two discrete surfaces. These calculations, intersection tests, etc., were performed using robust topological primitives that were implemented with a local tolerance that is specific to each calculation. The line segments forming the intersection were subsequently inserted into the intersecting meshes. Instead of inserting the edges globally in a three-dimensional mesh, the edges were inserted into each triangle individually. This process of local repair simplified the process of inserting the edges into the mesh by transforming the repair process into two dimensions. The subsequent node insertion and edge recovery are guaranteed in two dimensions. After all of the nodes and edges were successfully inserted into the mesh the intersection was considered repaired, that is, the intersection was removed by replacing it with nonmanifold edges.

A surface-painting algorithm that is bounded by surface boundary edges, free-boundary edges, and nonmanifold edges was used to 'break out' the surfaces that are bounded in whole or in part by the newly created nonmanifold edges. The nondesired geometry can then be removed to arrive at fully repaired geometry that is manifold and free of boundary edges at the intersection. This method repairs intersecting discrete geometry in-place, that is, without having to remesh the intersecting surfaces. Results show that these algorithms effectively, efficiently, and accurately capture and repair the intersection of discrete surfaces.

REFERENCES

1. Bischoff S, Kobbelt L. Structure preserving CAD model repair. *Eurographics* 2005; **24**(3):572–536.
2. Lo SH, Wang WX. Finite element mesh generation over intersecting curved surfaces by tracing of neighbors. *Finite Element in Analysis and Design* 2005; **41**:351–370.
3. Aftosmis MJ, Berger MJ, Melton JE. Robust and efficient cartesian mesh generation for component-based geometry, U.S. Air Force Wright Laboratory / NASA Ames, CA.
4. Samet H. *The Design and Analysis of Spatial data Structures*. Reading: Addison-Wesley Publishing: Boston, MA, 1990.
5. Park SC. Triangular mesh intersection, Department of Information & Systems Engineering, Ajou University, August 2004.
6. Möller T. A fast triangle-triangle intersection test. *Journal of Graphics Tools* 1997; **2**(2):25–30.
7. Gellert W, Gottwald S, Hellwich M, Kästner H, Knstner H, Weisstein EW (eds). *VNR Concise Encyclopedia of Mathematics*, 2nd ed. Van Nostrand Reinhold: New York, 1989. 541–543.
8. Edelsbrunner H, Mücke EP. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms, University of Illinois at Urbana-Champain, 1990.
9. Weisstein EW. Rotation matrix. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/RotationMatrix.html.
10. Lawson CL. Properties of n-dimensional triangulations. *Computer Aided Geometric Design* 1986; **3**(4):231–246.
11. Stanford 3D Scanning Repository. Graphics.stanford.edu/data/3Dscanrep.