8-7-2010

Algorithms and methods for discrete mesh repair

David Owen McLaurin

ALGORITHMS AND METHODS FOR DISCRETE MESH REPAIR

By

David Owen McLaurin

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Aerospace Engineering
in the Department of Aerospace Engineering

Mississippi State, Mississippi

August 2010

ALGORITHMS AND METHODS FOR DISCRETE MESH REPAIR

By

David Owen McLaurin

Approved:

David Marcum Professor, Mechanical Engineering Dissertation Director	Pasquale Cinnella Head of Aerospace Engineering Dept. Major Professor
Eric Blades Adjunct Faculty Committee Member	Mike Remotigue Associate Research Professor, CAVS Committee Member
David Thompson Associate Professor, Aerospace Engineering Committee Member	
Mark Janus Associate Professor, Aerospace Engineering Graduate Coordinator	Sarah A. Rajala Dean, Bagley College of Engineering

Name: David Owen McLaurin

Date of Degree: August 7, 2010

Institution: Mississippi State University

Major Field: Aerospace Engineering

Major Professor: Pasquale Cinnella, Ph.D.

Title of Study: ALGORITHMS AND METHODS FOR DISCRETE MESH REPAIR

Pages in Study: 140

Candidate for Degree of Doctor of Philosophy

Computational analysis and design has become a fundamental part of product research, development, and manufacture 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 mesh generation. This will be accomplished by developing tools that semi-automatically repair discrete data. Providing a 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 semi-automated solutions to complicated geometrical problems—specifically discrete mesh intersections and isolated boundaries.

The intersection-repair strategy presented here focuses on repairing the intersection in-place as opposed to re-discretizing 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 non-manifold, non-

intersecting geometry that is free of duplicate and degenerate geometry. Results are presented showing the accuracy and consistency of the intersection repair tool.

Isolated boundaries are a type of gap that current research does not address directly. They are defined by discrete boundary edges that are unable to be paired with nearby discrete boundary edges in order to fill the existing gap. In this research the method of repair seeks to fill the gap by extruding the isolated boundary along a defined vector so that it is topologically adjacent to a nearby surface. The outcome of the repair process is that the isolated boundaries no longer exist because the gap has been filled. Results are presented showing the precision of the edge projection and the advantage of edge splitting in the repair of isolated boundaries.

DEDICATION

This dissertation is dedicated to my parents, Bert and Daphne McLaurin, and my wife, Cheryl McLaurin, without whom this would not have been possible.

ACKNOWLEDGEMENTS

I deeply appreciate the infinite patience, help, and support throughout of Dr. Eric Blades, Dr. David Marcum, and Dr Mike Remotigue. I would also like to thank my dissertation committee members, Dr. Pasquale Cinnella and Dr. David Thompson, for their perspective and experience. In addition I would like to thank all of my professors in the College of Engineering, especially the Department of Aerospace Engineering, for the inspiration and thirst for knowledge they instilled in me during my studies. Also, thank you to all my friends for being great sounding boards through the years—particularly Bill Michaels, Brandon Witbeck, and Brett Ziegler. Finally I would like to especially thank my wonderful wife, Cheryl. She was always supportive and encouraging throughout my years of study.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Computational analysis and design has become a fundamental part of product research, development, and manufacture in aerospace, automotive, and other industries. The process typically begins with the construction of a computational model to use as a virtual representation of a 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.

Unfortunately, the creation of a computational model can be a difficult and time consuming task. The creation process includes geometry preparation, repair, clean-up and mesh generation and each of these steps can require a significant amount of man-time. If an application involves a complex geometry, the geometry preparation can be a time consuming. If a geometry has not been prepared for simulation purposes or exhibits errors that originate from numerical inaccuracies, then the geometry repair process can be time consuming. Also, the process of removing unwanted or unnecessary geometry can be a long process. In addition, most volume-mesh generators require that the discrete

input geometry be clean, i.e. watertight and manifold. A watertight discrete geometry is one that contains no free or boundary edges, and manifold discrete geometry contains no edges that have more than two topologically attached elements.

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 due to non-linear operations, such as intersections and projections using high-order non-uniform rational b-splines (NURBS). However, performing intersections or projections using discrete geometry involves linear operations using linear elements—at least for this research. 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. Some of the issues related to efficiently producing accurate and consistent computational models will be discussed in the next section.

Issues with Computational Models

The process of performing a computational simulation includes creating a computational (CAD) model, simulation, and post-processing the results from the simulation. 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 seventy-five percent of the man-time for an overall numerical simulation is spent during the geometry preparation, repair, clean-up, and mesh generation [1].

In recent years, many tools for repairing CAD models have been developed. Since the field of CAD model repair is so vast, the tools developed are usually quite specific in their application—only solving one problem in a certain number of cases. This method of progress in this field is justified since a developing a tool that is general enough to solve most problems has been impossible. This is primarily due to the infinite arrangements of geometry, relative scales of geometric components, and requirements of the geometry based on its intended application. However, many researchers have offered solutions for simple, ubiquitous problems whose repair can be automated [1],[2],[3],[4]. Some of the common problems that can be present in CAD models are gaps, overlaps, duplicate geometry, degenerate geometry, and intersecting geometry. Due to varying length scales present in CAD models and the lack of topology relations [5], problems such as these have to be repaired by using discrete geometry repair techniques which directly alter the topology through the addition or removal of elements or by gluing of edges or vertices. These operations can be performed manually; however methods or algorithms that offer some degree of automation have, in some cases, significantly reduced the amount of time needed to create watertight geometry.

The aim of this work is to reduce the man-time needed to prepare geometry for mesh generation. Much progress has been made in recent years towards applying simple tools to repair simple problems. However, this research seeks to accelerate the process of preparing geometry for mesh generation by offering solutions to complex problems that cannot, in general, be automated. This will be accomplished by developing tools that

semi-automatically repair discrete data. The aspect of these complex problems that restricts the extent of automation in the repair process is the reliable and accurate classification of not only where to apply the tools but what to do with the results. The most prevalent problems, small gaps and overlaps, can in most cases already be repaired automatically without user intervention. However, larger, more complex problems, such as large gaps and intersections, have been found to be difficult to repair automatically due to differing length scales that may be present in a model. For example, a gap that needs to be repaired could be the same relative size as a necessary feature present elsewhere in the model. Repairing the gap automatically could also inadvertently remove the necessary feature. Therefore, large gaps and intersections must be identified manually and repaired semi-automatically using tools that are developed here. Providing a level of automation to the process of repairing these large, complex problems in discrete data will significantly accelerate the grid generation process. It will also remove the need for most CAD cleanup since a watertight, manifold, discrete representation can be generated from non-watertight, non-manifold CAD geometry using the developed tools.

Contributions

In this work, algorithms were developed to repair CAD models that exhibited specific problems which are unable to be fully automated—specifically intersections and isolated boundaries. Techniques that re-mesh 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 remove intersections and isolated boundaries. The developed

algorithms are meant to offer semi-automated solutions to complicated geometrical problems. 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.

Repair of Discrete Mesh Intersections

Since 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 re-discretizing the areas found to intersect. In addition, current methods are usually restricted to geometries that are already watertight.

The work presented here focuses on repairing the intersection in-place as opposed to re-discretizing the intersecting geometries. This removes the potentially large expense associated with the re-discretization process and lifts the requirement of the input mesh being watertight. Repairing the mesh in-place entails directly altering the geometry topology. Intersections are detected through the use of an octree data structure, 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 through 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. This further reduces 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 performing 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 [6],[7],[8]. The result of this intersection process is a non-manifold, non-intersecting geometry that is free of duplicate and degenerate geometry. Some results demonstrating the various aspects of the tool will be presented in CHAPTER VII.

Repair of Isolated Boundaries

Isolated boundaries are a type of gap that current literature does not address directly. They are defined by discrete boundary edges that are unable to be paired with nearby discrete boundary edges in order to fill or repair the existing gap. In addition, these types of gaps are in general not able to be repaired by adapting existing techniques. Therefore, the tool developed here solves a problem that previously had no general solution. Other discrete repair techniques require the presence of a hole, a pair of boundaries, or use volumetric techniques. By definition, isolated boundaries cannot be paired since they are isolated, and cannot typically be reduced to a hole and then filled. Also, as stated before, volumetric techniques will not be used here due to the prohibitive computational cost and inability to retain small features in a discrete geometry.

In this research the method of repair seeks to fill the gap by extruding the isolated boundary along a defined vector so that it is topologically adjacent to a nearby surface. The method presented here begins by projecting the isolated boundaries onto the nearest

surface along a defined normal vector. The projection is accomplished by inserting the isolated boundary edges at the point where the defined normal vector pierces the nearest surface. The pierce points are calculated through the use of ray-casting within the octree data structure—which significantly reduces the number of ray-element intersection tests required to project the edges. At this point the edges could be wholly inserted, or recovered, into the nearby surface. However, a method of splitting the projected edge, which was found to preserve discrete element quality around the projection, was developed and is demonstrated to be superior to whole-edge projection and recovery in this application. The gap between the isolated boundary and the projected-upon surface is then filled by creating new discrete elements. The outcome of repair process is that the isolated boundaries no longer exist because the gap has been filled. Results demonstrating the various aspects of the tool will be presented in CHAPTER VII.

Supporting Data Structures

Although these contributions are not original, their combination and application as used here is unique—and essential to the success of this research. Because these processes are designed to be controlled by a user during the repair process, much effort has been made to reduce the time required to repair the selected portions of the model. The supporting data structures have been designed to be light-weight both computationally and on storage so they can be implemented inside a typical CAD or mesh generation system. This ensures that the procedures do not hinder the repair process through long repair times.

A hierarchical data structure was chosen to represent the mesh because of the flexibility with respect to building and maintaining mesh maps as well as the algorithmic

efficiency associated with the most commonly used mesh operations. Building and maintaining the mesh maps is done automatically when any topological entities, elements, vertices, or edges, are created or destroyed. All repair processes are defined at the fundamental level as element creation or destruction. Therefore, before and after any operation the integrity of the mesh maps is maintained and they do not have to be rebuilt at any point after their creation. This not only makes the repair processes efficient it also accelerates the process of software development by removing the responsibility of maintaining mesh-map integrity from the user.

The intersection and ray-casting routines present in this work rely on a spatial-subdivision data structure, the octree, to reduce the algorithmic complexity of finding intersections and projections from $O(n^2)$ to approaching $O(log(n))$. The octree is also used to increase the efficiency and accuracy of the edge projection routines. Finally, the octree itself was optimized for both memory requirements and query efficiency. This was accomplished by only splitting the octree in areas where the geometry is relatively dense, i.e. the octree is not resolved as finely in areas where very little geometry is present. This saves both space and makes the queries more efficient since there are fewer octants to query.

Organization of Dissertation

Chapter II presents background information and a literature review of the issues related to unstructured mesh repair. Chapter III and Chapter IV present the current implementations of supporting data structures that were essential to this research. The intersection repair algorithm is presented and explained in Chapter V. The isolated-boundary repair algorithm is presented and explained in Chapter VI. Chapter VII

presents results of both mesh repair algorithms developed in previous chapters. Finally a summary of contributions and possible future work are presented in Chapter VIII.

CHAPTER II

LITERATURE REVIEW

Introduction

The need for tools to repair both discrete data and CAD models is obvious. However, since their means of representing geometry are fundamentally different, the methods of repairing common problems also differ. To repair CAD models, there are two choices: repair the CAD definition itself using simple CAD operations or discretize the dirty CAD and repair the resulting discretization. Attempts to automate the former approach have had success with relatively simple geometries, but have had limited success with more complex geometries [5],[9],[10]. For reasons of robustness, the latter approach will be reviewed here.

Discrete data repair is done by changing the topology through the addition or subtraction of elements from the data or manipulation of existing elements to arrive at a desired result—in this case watertight, manifold geometry. Since most portable CAD model formats do not provide inherent topology information, it has to be derived by simple CAD operations such as projecting curves (NURBS), creating trimmed surfaces, and splitting/gluing surface edges, etc. [1]. In practice, the aforementioned simple CAD operations often are unable to be automated because of the varying length scales present in CAD models and the lack of topology relations [5]. For this reason, the current research will focus on the development of discrete repair techniques for discrete data and then the application of those techniques to mesh generation from dirty CAD models.

Discrete repair techniques can be generalized into at least three categories that will be considered here: hole-filling based, volumetric based, and vertex-pair based.

General Mesh Repair Techniques

Mesh Repair via Hole Filling

Hole-filling techniques are a good tool to have because they deal with a common problem. Many options are available to fill holes, from simply collapsing boundary elements to adding triangles with a unit disk. The interior geometry of the hole can even be extrapolated from the elements surrounding the hole in an effort to add curvature in the interior or to enforce a boundary condition around the hole. In general, gaps can often be made into holes, which can then be filled. Hole filling is a simple, robust technique to repair geometry and can be useful since many more complex problems can often be simplified to include the filling of a hole.

Holes in discrete data can arise from tolerance differences between CAD models and the mesh generator, missing information from using range scanners, etc. Holes are defined here as a set of connected free boundary edges that form a closed loop that do not define the perimeter of a surface. Since holes are a closed loop, it follows that the addition of elements to the discrete data to fill the hole is appropriate. An algorithm involving the numerical optimization scheme from probability calculus called simulated annealing was developed by Wagner, *et al*. [11]. This involves a preprocessing step (removing "bad" triangles around the perimeter of the hole) and the random changing of the topology (adding triangles and swapping edges) until all of the holes are closed.

J. P. Pernot, *et al*. [12] developed an algorithm that fills holes by first clearing the free boundary of undesirable triangles. Next, the hole is filled with the use of a unit disk,

which has the same number of boundary vertices as the hole boundary, that is placed at the centroid of the hole. Generating the new surface patch is now straightforward since the unit disk and the boundary of the hole have a one-to-one matching on vertex count. The new surface patch is given curvature by the addition of new vertices whose position is determined by a curvature-variation-minimization scheme. Another algorithm developed by Liepa [13] is based on element subdivision: A hole is originally triangulated using an extended version of an existing algorithm [14]. Then the interior is refined via edge swapping to approach a specified mesh quality. Finally, the interior is faired through element subdivision to shape the newly generated elements to match the surrounding mesh. A similar approach is taken by Levin [15]: the hole is filled using quadrilateral elements that are then subdivided based on free boundary conditions placed on the edges that define the hole.

Finally, a more general algorithm was developed by Guo, *et al*. [16] that addresses general gaps that do not have to be closed. Voxel diffusion was used to advance the boundaries of the surfaces into space until collisions were detected. The gaps were then filled with triangles and refined to a specified limit. Curvature was given to the new geometry in the hole during the diffusion process by using the Marching Cubes Algorithm [17].

Hole filling and, more generally, gap filling seeks to repair discrete data by generating new elements to fill the hole/gap. In current research these methods have been applied to small gaps instead of the large ones that this research aims to repair. However, generalizing the fundamental components of small gap filling and then applying them to large gap seems to be a promising approach to generating a solution.

Volumetric Techniques for Mesh Reconstruction

Another rich field is mesh reconstruction using volumetric techniques. These completely rebuild the input mesh and produce a guaranteed watertight, manifold triangulation. Two common volumetric techniques are either to use a space filling data structure, such as an octree or k-d tree [18], or to convert the surfaces to a signed distance function [19]. The data structure or distance function is then resolved to a tolerance, ε, away from the original geometry. This process generally results in a loss of sharp features since the conversion from mesh data to volumetric representation and back acts as a low-pass filter [1]. In this case, the "low frequencies" that are allowed to pass through the conversion process, which acts as the filter, are the large features in the model. The sharp features in the model are treated as "high frequencies" by the conversion filter and can be lost. The resulting mesh from this technique can also be overly tessellated to a large degree [1] and may need to be decimated [20]. Bischoff, *et al*. [21] developed an algorithm that uses an octree that is intersected with the input data to produce watertight, manifold model. The octree used in the aforementioned algorithm is completely resolved to the tolerance ε, which can be very memory intensive.

Bischoff, *et al*. [1] later developed an algorithm that uses an octree to satisfy the tolerance ε locally by identifying "critical grid vertices" and applying the volumetric repair techniques to only these areas. It is stated that this does not have the same performance penalties associated with globally reconstructing the data. Volumetric techniques are attractive from an automated mesh repair point of view because they are guaranteed to produce clean geometry in one step that is a finished mesh. However, these techniques can destroy most of the structure of the discrete data from the global resampling inherent in this method. Experimental results, [1], [21], show that mesh

repair via volumetric techniques can also be a time-consuming and resource intensive process.

Vertex of Edge Based Mesh Repair

Next is the general field of repairing a mesh by modifying the geometry and topology directly. These methods do not assume the presence of a closed hole, only that the model has free boundary edges that have to be repaired. Many different names are applied to the operations that either glues free boundary vertices together or fills in the gap between them. Patel, *et al*. [2] labeled them "stitching" and "filling" respectively, and developed algorithms that applied a distance-based tolerance to repairing dirty meshes. Any free boundary vertices that do not belong to the same surface are paired if the distance between them is below a tolerance α. A glue tolerance, γ, is used to determine if the pair was to be glued together or the space between them filled with the addition of elements.

Chong, *et al*. [3] improved upon these simple operations and applied them to more complex geometrical problems. Vertex pairs are still used as the basis for repairing the geometry, but if the gap needs to be filled with elements, a new operation is used. The gap is turned into a hole by bridging the gap with elements which changes the topology to create a simply-connected region, as illustrated in Figure 2.1.

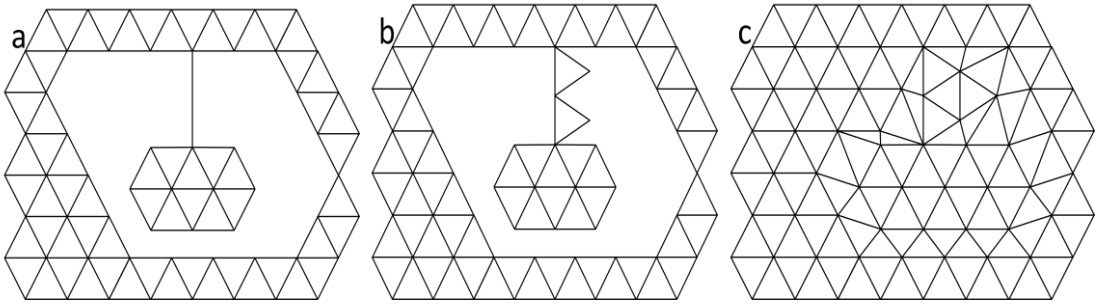


Figure 2.1 (a) Filling a ring hole by bridging two loops; (b) Creating elements on the bridge; (c) Filling the ring hole with elements

Then a hole filling technique similar to those discussed in earlier is used to repair the geometry. A feature that Chong [3] and Patel [2] use to make the gluing process more robust is the addition of edge splitting. If the edge of a triangle attached to a paired vertex, *a*, is closer than the complimentary vertex in the pair, *b*, then the edge is split by projecting the vertex, *b*, onto the edge and forming a new pair with the new vertex as seen in Figure 2.2 and Figure 2.3.

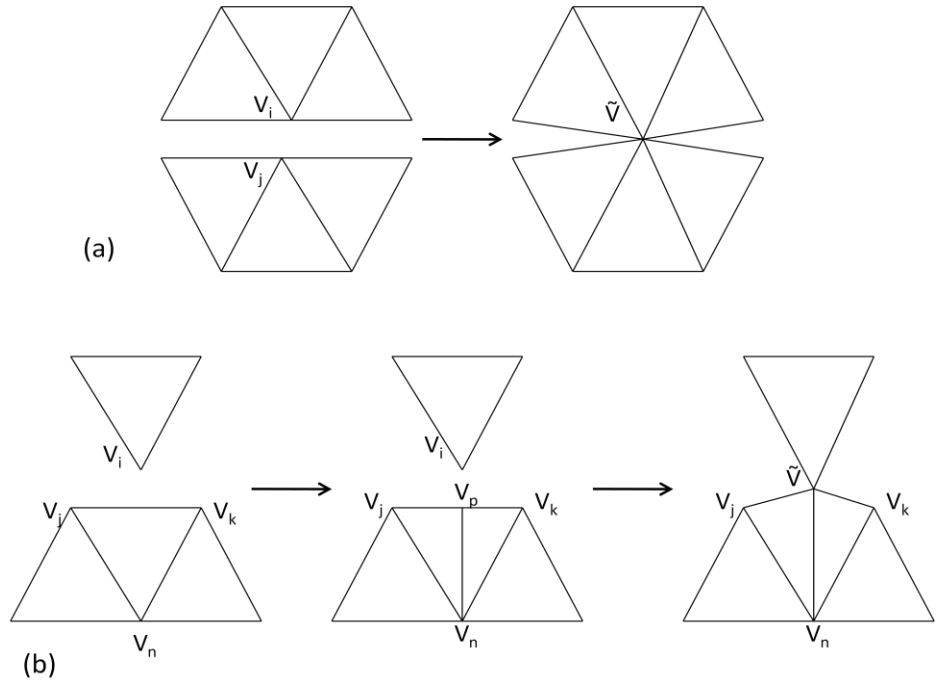


Figure 2.2 Vertex-Pair contraction operation: (a) without edge split operation; (b) with edge split operation

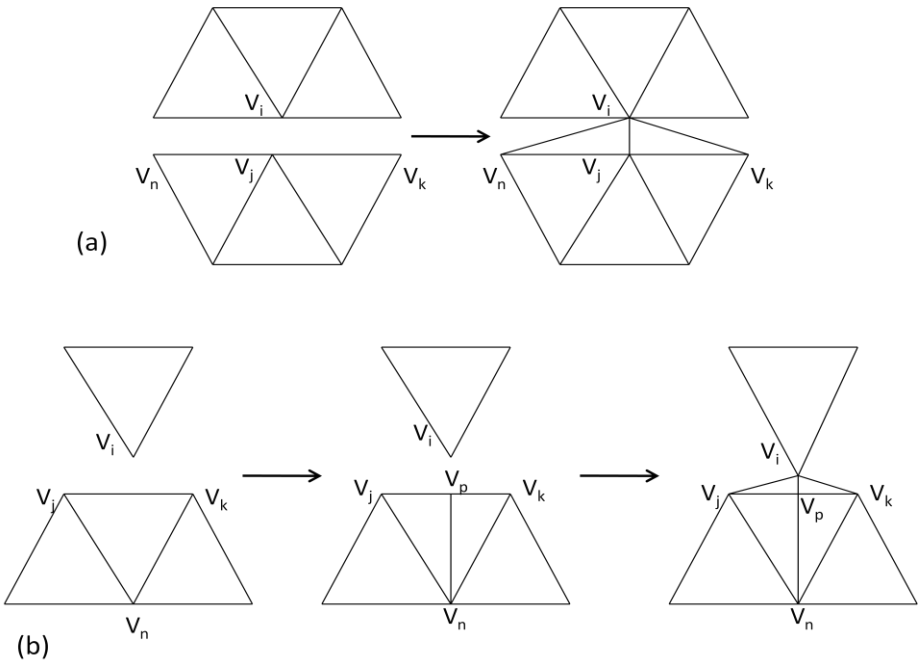


Figure 2.3 Vertex-Pair expansion operation: (a) without edge-split operation; (b) with edge-split operation

Edge splitting operations lessen the possibility that many skewed elements could be formed or many triangles could be removed due to edge collapse. The advantage of these vertex-pair based methods is the low performance overhead associated with these relatively simple operations. Because of their simplicity, these methods are hard to apply to problems that are more complex. However, these methods can form a solid basis from which to develop more complicated tools.

Specific Mesh Problems

Intersecting Mesh

In many cases, the geometrical problems present in data do not fit in the above categories (simple holes, small gaps, and small overlaps). Intersecting elements can be present in geometry that is completely valid and watertight as shown in Figure 2.4.

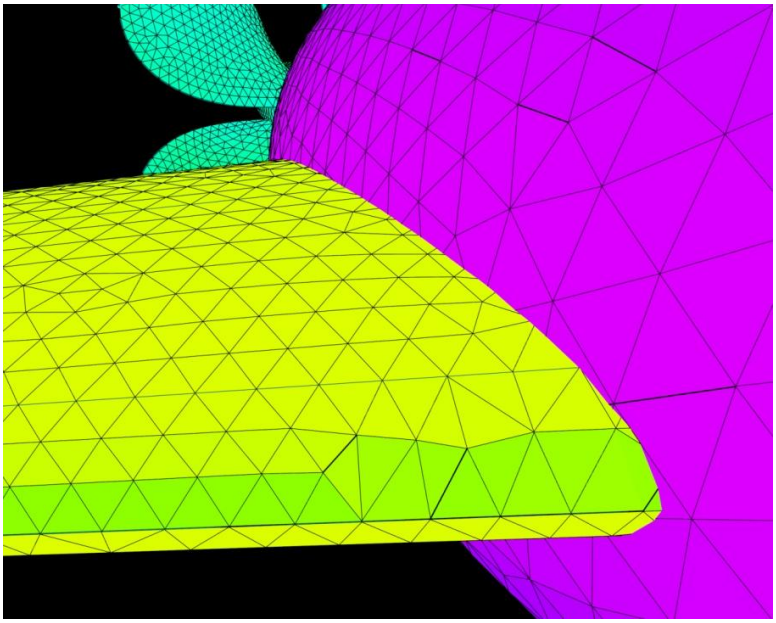


Figure 2.4 Mesh intersection between right wing (yellow) and fuselage (purple)

Many researchers have presented robust and efficient intersection tests for discrete data elements [6], [7], [8], [22]. For example, Cartesian mesh generation necessarily involves the intersection of the regular Cartesian grid with the existing free boundary elements or edges. However, the implementation of repairing the intersection is very sensitive to round-off error and is therefore difficult to develop and implement. In addition, if some method to reduce the number of intersection tests is not employed, then the process of searching for intersections becomes an $O(n^2)$ operation in the number of triangles.

Park, *et al.* [6] developed a method to separate the data into groups that are not self-intersecting using visibility maps. This method drastically reduces the frequency of computationally expensive triangle-triangle intersection tests when compared to the $O(n^2)$, brute-force method. Aftosmis, *et al*. [7] developed a three-dimensional Cartesian mesh generator that featured an intersection routine of generally positioned triangles whose aim was to combat the negative effects of round-off error. The intersection tests that were developed involved only multiplication and addition and were found to be robust. The effects of round-off error were diminished through the use of exact arithmetic, when necessary, and the establishment of tie-breaking routines for degeneracies using virtual perturbations. Lo, *et al*. [8] presented an algorithm for generating finite element meshes from intersecting curved surfaces. The number of triangle intersection tests is reduced by the use of "neighbor tracing." "Neighbor tracing" involves attempting to create a chain of intersecting edges through intersection tests of topological neighbors of intersecting triangles. The chain of edges was then used to re-mesh the intersecting geometry. It must be noted that the current research referenced above began from watertight, manifold geometry and since the wetted area of intersecting geometries was the goal of the research, the geometry "inside" the model could be

removed automatically. This research has the goal of repairing intersecting geometry in-place instead of re-meshing. Also, the input geometry need not be watertight or manifold.

Isolated Boundary

The other problem to be addressed here is repairing a gap or overlap that does not have a pair of boundaries (isolated boundary). No current research was found that directly addresses this problem. The general repair techniques discussed above could be used in certain situations, but a more general solution must be developed. A brief explanation of the problem follows: In Figure 2.5 the left wing (pink) of a discrete model has an obvious gap between it and the fuselage (purple). The yellow lines in Figure 2.5 represent free boundary edges. As seen in Figure 2.6, the fuselage has no free boundary edges near the left wing, which is floating in space near the fuselage. Vertex-pair based algorithms are unable to fill this gap since only one of the components has free boundary edges. Hole filling algorithms are also unable to repair this problem because they would just place a cap on the end of the wing. This would remove the free boundary edges but not fix the problem.

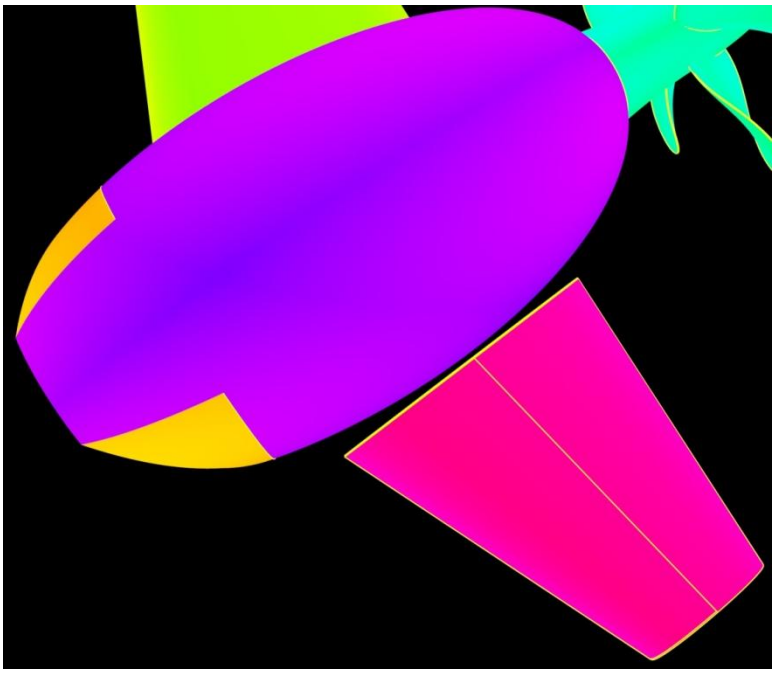


Figure 2.5 Discrete model with gap present between fuselage (purple) and left wing (pink)

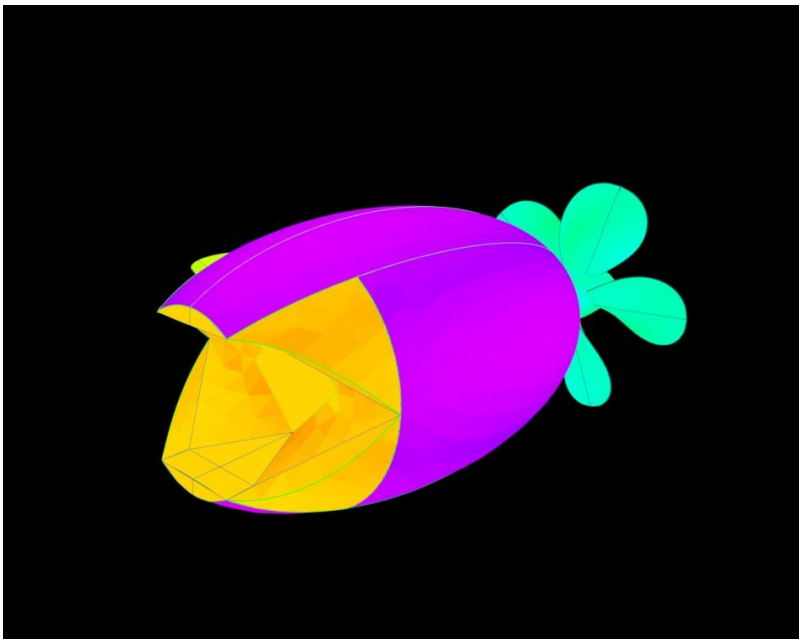


Figure 2.6 Discrete model from the left side showing the fuselage with no boundary edges

Conceptually, the solution is simple. If this were a CAD model, it could be repaired by projecting the NURBS-curves onto the fuselage (NURBS-surface), trimming the fuselage surface, filling the gap between the wing and fuselage, and then deleting the unnecessary fuselage surface inside of the trimming loop. However, the associated discrete operation of projecting edges onto discrete data is sensitive to round-off error similar to self-intersecting meshes. Once the edges are projected, the gap between the wing and fuselage could be filled with triangles using any of the vertex-pair based repair techniques. However, non-manifold geometry has now been created and must be repaired. If the projected edges form a closed loop, then the region "inside" the projected edges can be identified and removed to form manifold geometry. However, if the edges do not form a closed loop, then nothing further can be done because the concept of "inside" and "outside" cannot be applied.

CHAPTER III

MESH DATA STRUCTURE

Introduction and Brief Overview

The selection of a data structure in any computational project is of utmost importance, as the algorithmic complexity of a given algorithm can be made worse simply because of a poor choice. Therefore, the selection should be delayed until it is determined what will be required of the data structure. In the case of mesh repair, data structures often take on one of two forms: array based (contiguous memory) and list based (non-contiguous memory). Array based implementations are generally used when the primary use of the data structure is element access, which for arrays is O(1). List based implementations are generally used when the primary use of the data structure is element insertion and removal, which for lists is O(1). Even though this is a simple generalization, it serves as the basis for the fundamental choice of data structure in this research.

Representing the to-be-repaired mesh is not the only requirement of the chosen data structure. Since the represented mesh will necessarily be changing throughout the repair process, adding and removing elements must be considered. Adding and removing elements from both an array and a list can be made to be an amortized O(1) operation. With the use of a system that can flag elements in an array as invalid, the need to remove the data from an array can be delayed. Therefore, the addition of elements only need be on an end. This operation, along with the addition of clever timing and size of

reallocation of an array, makes element addition and removal an amortized O(1) operation. Inserting and removing elements from a list is also a O(1) operation. Therefore, either data structure could achieve the optimal algorithmic complexity of O(1) for adding and removing elements.

A mesh is classified as non-manifold if an edge in the mesh has more than two elements attached. Both array-based mesh representations and list-based mesh representation have no fundamental problems representing non-manifold meshes. The problem comes when contemplating how to create and represent mesh maps—which support adjacency queries.

Adjacency queries are also required and are perhaps the most demanding requirement of the data structure. Adjacency queries are achieved through the use of maps. To query all of the elements that are topologically adjacent to a vertex, a map from vertices to elements is needed. To query the elements that are topologically adjacent to an element, a map from elements to elements is needed. In addition, the integrity of these maps must be maintained since generating them for the entire mesh for every query is impractical. Generating the maps should ideally only be done once, and the incremental maintenance can be very expensive if proper care is not taken in the representation of the maps.

The subject of mesh data structures and maps that support associated queries is well documented. Therefore, only the conclusion that was reached after research will be presented. A hierarchical data structure was chosen to represent the mesh as it leads to a straightforward representation of the maps and supports non-manifold meshes. Nodes have a list of topologically attached edges and a list of topologically attached elements. Edges are defined by two nodes and have a list of attached elements. Elements are

defined by a list of nodes and a list of edges. These lists are filled during mesh creation and destruction. By defining mesh operations through the use of element creation and destruction instead of directly altering the data structures, the integrity of the maps is guaranteed before and after all mesh operations.

Hierarchical Mesh Data Structure Implementation

A hierarchical mesh data structure works by defining mesh entities, e.g. nodes, edges, and elements, in a hierarchy of increasing complexity. Nodes are at the bottom of the hierarchy as they are defined by a three-dimensional coordinate. Edges are next up in the hierarchy and are defined by two nodes. Two-dimensional elements are next up in the hierarchy and are defined by a list of nodes and a list of elements. Volume elements are not implemented in this research but could be by defining volume elements by a list of nodes, a list of edges, and a list of facet elements. Note that the specific facet element does not matter for mesh representation. Figure 3.1 is a graphical representation of the mesh hierarchy. Red arrows represent a hierarchy direction and black arrows represent mesh maps. This mesh in this research is implemented as an exclusively triangular mesh. A mixed element mesh could be supported with the limitation that all modified or new elements will be triangles.

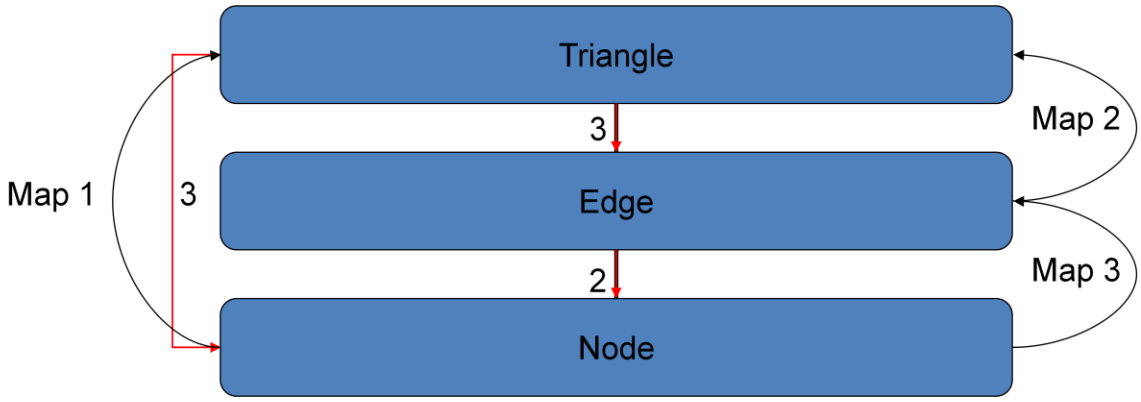


Figure 3.1 Hierarchical Mesh Data Structure

Hierarchical data structures also lend themselves well to mesh maps. For instance, classical mesh maps, which are used to represent manifold meshes, store adjacency information from element to element. This becomes complicated if the mesh is non-manifold because determining which edge to cross to get to an adjacent element becomes non-trivial. However, through the use of hierarchical data structures element adjacency for non-manifold meshes is simplified through the use of a hierarchy. Instead of creating map of adjacent elements for each element, a map of elements that are topologically adjacent to individual edges is created. This is then used to determine the elements that are topologically adjacent to each element. Non-manifold meshes pose no problem to this representation of mesh maps.

Mesh Maps and Queries

Through the use of a hierarchical data structure, hierarchical maps can be created and maintained in a straightforward manner. At the bottom of the hierarchy, the node, a map that contains the topologically attached elements for each level of entity above is created. The node class has two explicit maps: a list of topologically attached edges,

Map 3, and a list of topologically attached elements, Map 1. At the edge level of the hierarchy, one explicit and one implicit map exist. The edge class has a list of topologically attached elements, Map 2. Implicitly, the map of nodes that are topologically attached to the edge is present since two nodes define an edge. At the element level of the hierarchy, two implicit maps exist. Implicitly the map of nodes that are topologically attached to the element is present since the element is defined by a list of nodes. Also, the map of edges that are topologically adjacent to the element is present since the element is defined by a list of edges. Note that an explicit map that defines the elements that are topologically adjacent is not defined. However, this map is defined implicitly since each edge that defines an element each have a list of topologically adjacent elements.

1. Edges attached to a Node: This map is defined for each node independently and is updated whenever an edge is created or destroyed.

2. Elements attached to a Node: This map is defined for each node independently and is updated whenever an element is created or destroyed.

3. Elements attached to an Edge: This map is defined for each edge independently and is updated whenever an element is created or destroyed.

4. Element Neighbors: This map is not defined explicitly but can be inferred most efficiently from either the elements-attached-to-a-node map, or the elements-attached-to-an-edge map.

Mesh Operations

Since the mesh and all maps are presumed valid before the mesh operations, the mesh operations should not violate that condition. These operations are all defined so that they will not invalidate any pointers or make any mesh maps incorrect. The following operations are used in the research presented here to repair meshes.

1. Creating Node: Creating a node does not involve the construction or update of any maps since it is the fundamental data structure. However, a node would not usually created without subsequently creating an edge(s) or element(s).

2. Creating Edge: Creating an edge requires two existing nodes and this operation adds the newly created edges to the edges-attached-to-a-node map for each of the nodes.

3. Creating Element: Creating an element requires at least three existing nodes. The edges required might exist in the mesh but will be created if needed. The element will be inserted in the elements-attached-to-a-node map for each node and the elements-attached-to-an-edge map for each edge.

4. Destroying Node: A node can only be destroyed if it is not topologically adjacent to any edges or any elements—that is no edges or elements are defined by the to-be-destroyed node. Otherwise, the edges and elements that are topologically adjacent to the node will have corrupted maps. Therefore, destruction of a node involves destruction of attached entities, edges and elements, to maintain map integrity.

5. Destroying Edge: An edge can only be destroyed if it is not topologically adjacent to any elements. Otherwise, the elements that are topologically adjacent to the edge will have corrupted maps—that is no elements are defined by the to-be-destroyed edge. Therefore, destruction of an edge involves destruction of attached elements. If the edge is not topologically adjacent to any elements, then the edge only needs to be removed from the maps of the defining nodes before being deleted.

6. Destroying Element: In the current implementation, only facet elements are supported. When destroying an element, first remove the element from each map

of each defining entity: remove element from elements-attached-to-an-edge map, and remove element from elements-attached-to-a-node map. Then check the lower dimensional entities to see if they should be destroyed. If any of the defining edges have an empty list of topologically adjacent elements, it should be deleted. If any of the defining nodes have an empty list of topologically adjacent elements, it should be deleted.

7. Gluing Nodes: Gluing nodes can be accomplished by destroying the elements attached to the from-node and creating new elements with the from-node replaced with the to-node. Care must be taken to create the new geometry before destroying the old geometry. Since orphaned nodes and edges will be destroyed automatically when elements are destroyed, creating the new geometry first ensures that all of the nodes needed by the new geometry will be present.

8. Collapsing Edge: Edge collapse can be seen as gluing two nodes together. However, the elements that are topologically attached to the to-be-attached have to be destroyed before gluing the nodes.

9. Surface Painting: Surface painting is an algorithm for using existing topology to extract portions of a mesh satisfying a certain criterion. For example, if a portion of a mesh is non-manifold, one could specify that the painting algorithm not travel across non-manifold edges. This would effectively "break out" portions of the mesh that are exclusively bounded by non-manifold edges. A queue data structure is used as the fundamental data structure for this algorithm. Let the queue be the SEARCHING_STRUCTURE, and any container can represent the STORING_STRUCTURE, which will contain the desired elements. Let *top()* be the next element in the STORING_STRUCTURE, and *pop()* be a function that removes the element first in the STORING_STRUCTURE. Finally, let EDGE_TEST() be a predicate that takes an edge and returns true if it can be travelled across, and false if it cannot. The painting algorithm is as follows:

    1. Supply seed_element
    2. SEARCHING_STRUCTURE.push(seed_element)
    3. while(SEARCHING_STRUCTURE is not empty)
        a. current_element = SEARCHING_STRUCTURE.top()
        b. SEARCHING_STRUCTURE.pop()
        c. mark_visited(current_element)
        d. loop through list of defining_edges_
            i. EDGE_TEST(current_edge) == true
            ii. if *element* on other side of edge is not visited
            iii. SEARCHING_STRUCTURE.push(*element*)

e. STORING_STRUCTURE.push(currernt_element)

The result of the painting algorithm is a fully populated STORING_STRUCTURE. The STORING_STRUCTURE is populated with elements that are bounded by edges for who EDGE_TEST() == false.

CHAPTER IV

OCTREE DATA STRUCTURE

Static Resolution vs Dynamic Resolution Octree

The purpose of an octree is to reduce the algorithmic complexity of spatial queries, e.g. nearest-node search and element intersection candidates [18]. One way to implement an octree is to refine the whole tree to a certain depth based on the desired number of nodes in an octant. In practice this can lead to a large number of empty octants. The current implementation of an octree attempts to increase the efficiency of the search by reducing the total number of octants. Instead of splitting the whole tree to a certain depth, only the octants that have more than the desired number of nodes are split. This means that areas in the tree that do not have very many nodes are not resolved fully and there are far fewer terminal or leaf octants. By reducing the number of octants that needs to be searched, the octree occupies less memory and is possibly more efficient.

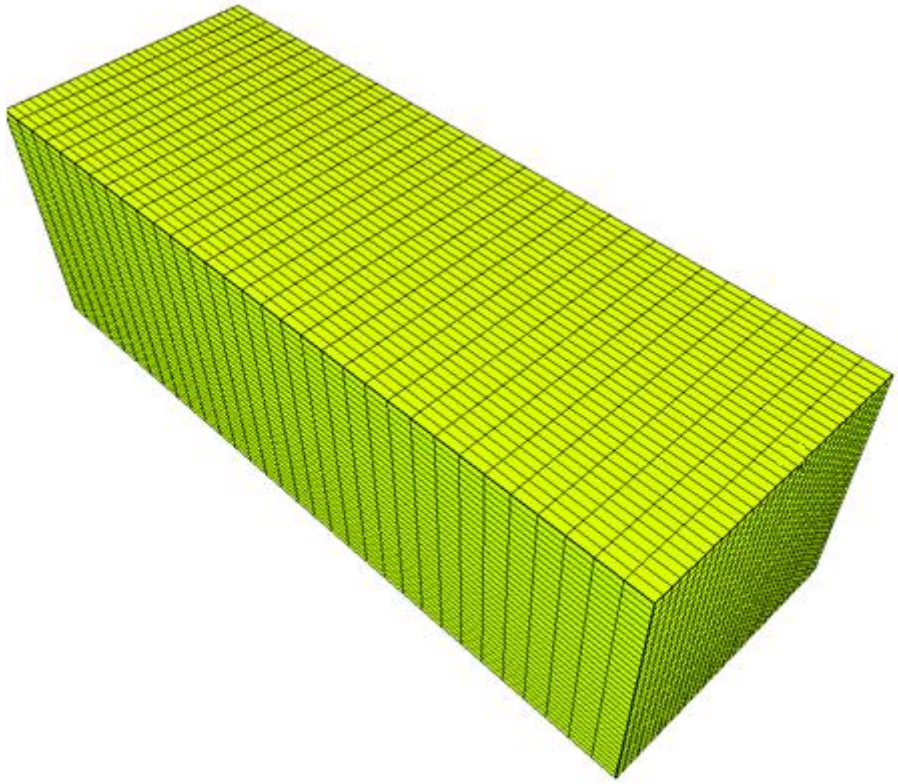


Figure 4.1 Uniformly Split Octree

The octree shown in Figure 4.1 is a uniformly split octree. This octree is split down to five levels; therefore it contains 8^5 or 32768 octants. It is constructed around half of a grill of an automobile. The domain has been normalized to the longest dimension. The longest dimension has length 1 and the other two dimensions have length 0.25. So the octree in Figure 4.1 is resolved down to $0.25 / 2^5$ or 0.0078125. Certainly this octree is fully capable of performing all of the tasks required; however, the ratio of storage requirement to number of empty octants is high because of the large number of empty octants.

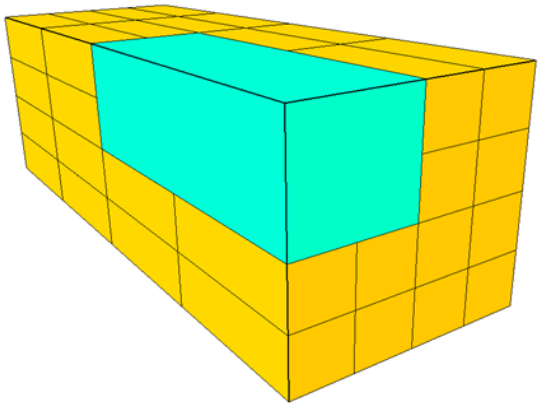


Figure 4.2 Non-Uniformly Split Octree

Consider the octree shown in Figure 4.2. It was formed around the same geometry but was not split uniformly. Instead octants were split only when an octant contained more than 5 nodes. The result is an octree with 64,996 octants with 12 levels. A similarly refined, uniformly split octree would require $O(10^{10})$ octants. The significance of the different colors is that each level was rendered is rendered in a unique color.

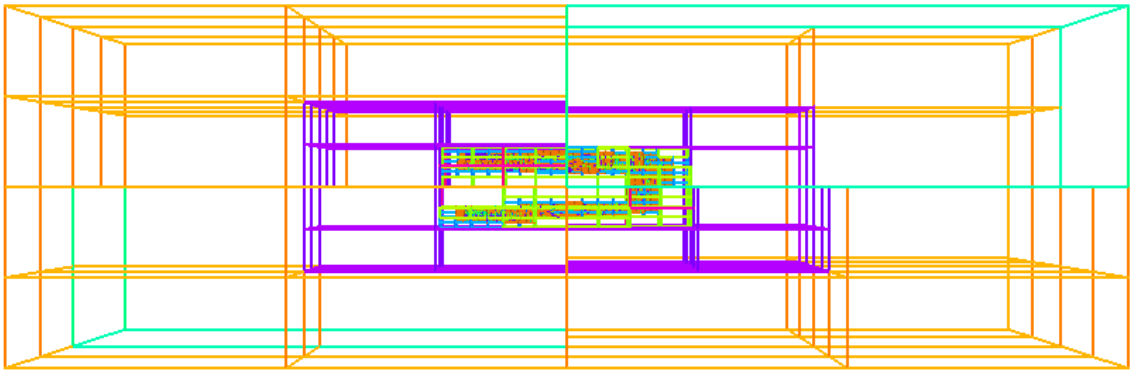


Figure 4.3 Non-Uniformly Split Octree, Wire Frame

Figure 4.3 shows the non-uniformly split octree rendered with a wire-frame instead of wire-fill so that the non-uniformity of the splits can be seen more easily. It is easily seen that there are far fewer octants in areas where no geometry is present. The octree is denser where the geometry resides.

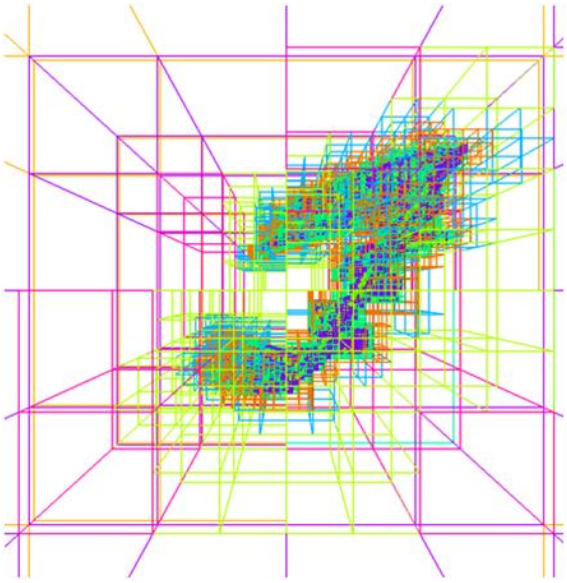


Figure 4.4 Close-up of end-on view of Non-Uniformly Split Octree

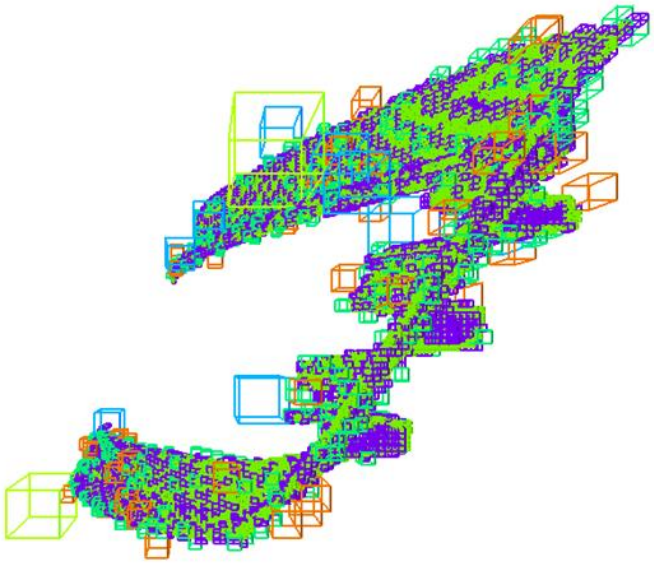


Figure 4.5 Close-up of end-on view of terminal octants in Non-Uniformly Split Octree

In Figure 4.4 an end of view of the whole tree is shown. The view is zoomed into the tree so that the geometry of the non-uniform splits can be seen. In Figure 4.5 only the terminal octants, those who have no children, are shown. It is easily seen that the octree has been split based on point density and it closely matches the geometry of the half-grill. Figure 4.6 and Figure 4.7 show the same octree from the front of the grill.

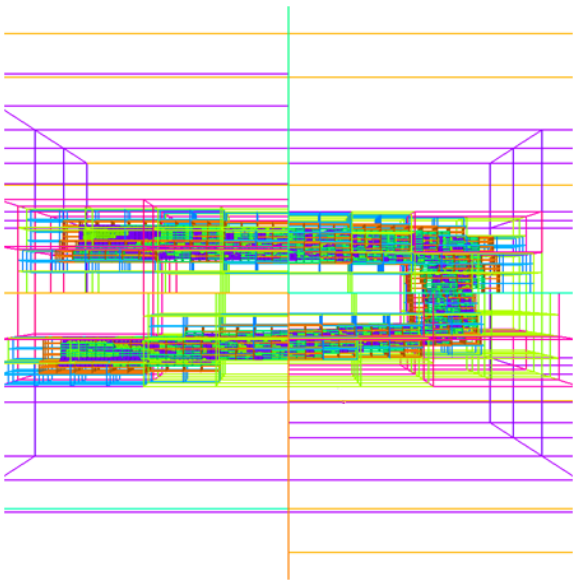


Figure 4.6 Close-up of front view of Non-Uniformly Split Octree

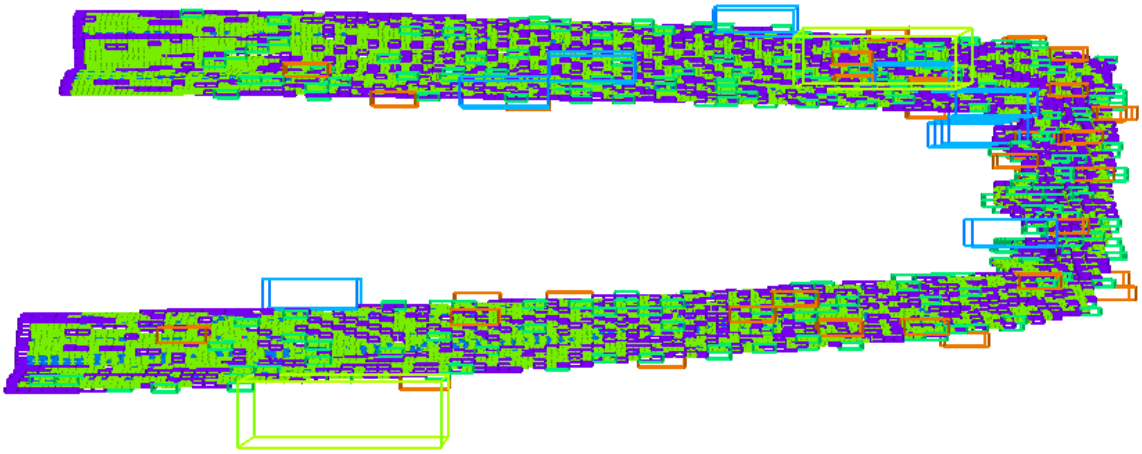


Figure 4.7 Close-up of front view of terminal octants of Non-Uniformly Split Octree

The dynamically split octree contains 64996 octants and 12 levels. The uniformly split octree contains 32768 octants. The dynamically split octree obviously contains about twice as many octants. However, the dynamically split octree resolves the space around the geometry to a much smaller level than the statically split octree. At 12 levels

of splits, the dynamically split octree has a minimum dimension of 0.25 / 2^{12} or 0.0006103. The dynamically split octree resolves the space around the octree two orders of magnitude more than the statically split octree. In order for the statically split octree to represent the same minimum dimension as the dynamically split octree the statically split octree would contain 8^{12} or 68,719,476,736 octants. That is a decrease of five orders of magnitude, ~10^5 octants for the dynamically split octree and ~10^{10} octants for the statically split octree, in the number of octants for the dynamically split octree.

If the statically split octree is a pointer-based octree, i.e. each level has 8^n octants, then the case against using statically split octrees grows worse. An octree split 5 times in each dimension has 8^0 or 1 octant on the zeroth level, 8^1 or 8 octants on the first level, 8^2 or 64 octants on the second level and so on for a total of 37,449 octants. The growth of the number of octants can be expressed by Equation 1. Splitting the octree one more level would require the 37,449 octants on the previous levels and would result in an octree with 299,593 octants. If split down to 12 levels, like the dynamically split octree, the number of octants required sky-rockets to 78,536,544,841.

$$\sum_{n=0}^{split\_level} 8^n \qquad \text{Eq. 1}$$

The worst-case storage requirements for a dynamically split octree approaches that of the statically split octree. The goal of this addition to the octree was to decrease the storage footprint for the octree. Reducing the storage requirements for the octree reduces the overall storage requirements for the supporting data structure. And, reducing the storage requirements for the supporting data structures frees memory for other uses,

e.g. repairing larger meshes. In terms of efficiency, the octree with fewer octants can only perform better than the octree with a greater number of octants, simply because there are fewer octants to search. Again, the worst-case performance in terms of number of octants searched approaches that of a statically split octree.

Octree Searching

Octree searches can be implemented with recursion. However, simulated recursion with the use of a stack or queue data structure can be used. A depth first search can be made using a stack to store visited octants. A breadth first search can be made using a queue to store visited octants. The stack data structure is used in this implementation because of the C++ implementation was faster. In fact, recursion is not present in this implementation of the octree. Whether a stack or a queue is used, the following algorithm is used here to search the octree. SEARCHING_STRUCTURE will be used in the algorithm listing to represent either data structure. The front of the queue and the top of the stack will be denoted as *top()*. A test for which octants to return or check is needed; in the following algorithm this will be stated as OCTANT_TEST. The container used to store the desired octants will be denoted as STORING_STRUCTURE.

1. Push root of octree into the SEARCHING_STRUCTURE.
2. While SEARCHING_STRUCTURE is not empty
    a. Current octant = SEARCHING_STRUCTURE.top()
    b. SEARCHING_STRUCTURE.pop()
    c. For each child of current_octant—skipped if octant is terminal
        i. If OCTANT_TEST(child) == true
            1. Push child on SEARCHING_STRUCTURE
        ii. Else
            1. Do nothing
    d. If current_octant is terminal
        i. Push current_octant on STORING_STRUCTURE
3. Return STORING_STRUCTURE

Storing Nodes in Octree

In order to insert a node, start at the top of the tree and determine which child would contain the node. Move to that octant, and repeat until a terminal octant is encountered. Put the node into the node bin of the octant. If the number of nodes in the octant is more than the threshold, then split the octant and distribute the contents of the recently split octant among the children of the octant. This method forms the tree around the geometry for optimal searches.

In order to remove a node from the Octree, find the containing octant with the above algorithm for finding the containing octant of a point. Remove the node from the node bin of the octant. If the number of nodes contained in the parent octant falls below the threshold for octant splitting then it needs to be consolidated into one octant with the nodes that are in the children. This method keeps the number of empty terminal octants minimized by consolidating children octants when the split no longer serves a purpose.

Storing Elements in Octree

One way to store elements in an octree is to store the elements in the smallest octant that contains all of the nodes. Since the larger octants will usually intersect a large number of elements, many elements will be stored in the largest octant, or root, or the tree. A query for intersection would involve returning all of the elements contained in an octant, both below and above the octant in the tree. Couple the method of storage with the method of querying and the tree no longer performs as $O(log(n))$; it would approach $O(n^2)$.

Another way to store elements is in terminal octants that intersect the element or more simply the bounding box around the element. A query for intersection candidates will to return the elements that are stored in the octants that intersect the bounding box of an element. This storage and query method performs much better and is implemented here instead of storing elements in the smallest octant containing all of the nodes.

Queries

Near Node List

This query will return a list of nodes that are within a tolerance given to the searching routine. The OCTANT_TEST for this query involves constructing a bounding box with the tolerance and testing for overlap between octants and the bounding box defined with the tolerance around the node. The nodes stored in the octants in the STORING_STRUCTURE returned from the query are then tested to determine if they are within the given tolerance. The list of nodes that passes this test is then returned from the octree.

Nearest Node

This query will return the closest node to a given node. Note, if nodes are equidistant from the queried node, the first one in the list is returned. This query is a near node query with the addition of a step that returns the closest node in the list of nodes returned from a near node list query. The tolerance used is first set as the longest dimension of the octant that contains the queried node. If nothing is found within that tolerance, the tolerance is doubled until either something is returned, or the search fails because the queried node is the only one in the tree.

Element Intersection Candidates

Element queries from the current implementation of the octree only return candidates that potentially pass a test, e.g. element intersection. Element-intersection-candidates query returns the elements that might intersect the query element. The OCTANT_TEST for this query involves constructing a bounding box around the element and testing for overlap between octants and the bounding box around the element. A list of elements that intersect the octants returned in the STORING_STRUCTURE from the octree search is returned as intersection candidates from the octree.

Ray Intersection Candidates

Ray-intersection-candidates query returns the elements that might intersect the query ray, or edge. The test for intersection (OCTANT_TEST) includes testing the six sides of each octant to determine if the ray pierces. It also includes a test to determine if the ray originates in the octant. A list of elements that intersect the octants returned in the STORING_STRUCTURE from the octree search is returned as intersection candidates from the octree. The ray-octant intersection routine implemented here was developed by [23] and the source code for this can be found in APPENDIX A.4.

CHAPTER V

MESH INTERSECTION

Introduction and Brief Overview

As stated in Chapter CHAPTER II, intersecting discrete geometry can be difficult to repair if care is not taken to effectively and efficiently lessen or remove the effects of round-off and truncation error. Also, some method of reducing the algorithmic complexity of testing for intersections from $O(n^2)$ must be developed. The repaired geometry must also be a valid mesh free of degenerate and duplicate geometry. Repairing intersecting triangles is a rich topic with much research [6],[22],[7],[8]; however the method implemented here for repairing intersecting, discrete meshes in place (without re-meshing) does not appear in the related literature. This section is organized as follows: discussion of triangle-triangle intersection tests, application of neighbor tracing to this problem, and local repair process.

Triangle Intersection Tests

Two triangle-triangle intersection (TTI) tests were found to be the most widely used in relevant literature [22],[7] and therefore will be discussed here. For the purposes of describing TTI tests, let us denote the two triangles T_0 and T_1, and the nodes of the T_0 and T_1 as N_{00}, N_{10}, N_{20}, and N_{01}, N_{11}, N_{21}, 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.

One type of TTI, first developed by Moller [22] and used by Lo and Wang [8], 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 [24]. 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. An example of this can be seen in Figure 5.1. On the left, the intervals along line L overlap; therefore, the triangles intersect. On the right, the intervals do not overlap; therefore, the triangles do not intersect.

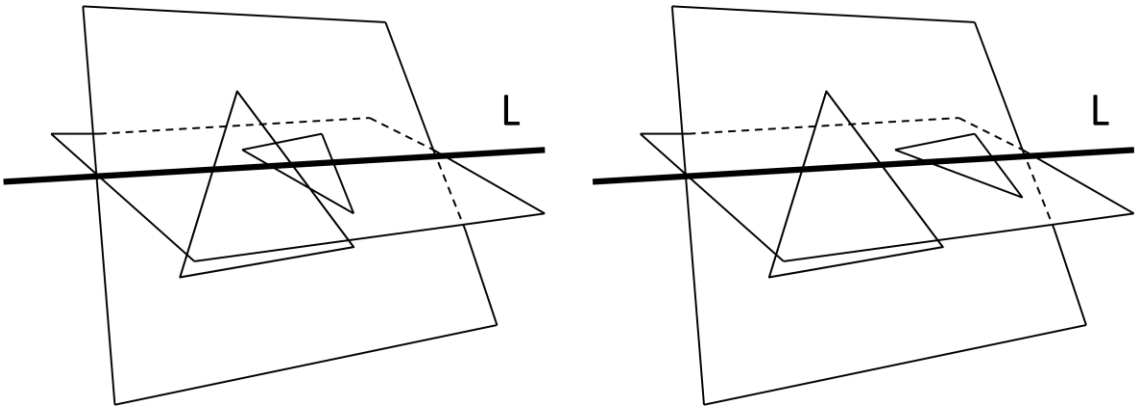


Figure 5.1 Triangle/plane intersection, with intersection line overlap(left) and without intersection line overlap(right)

Moller [22] developed optimizations for this method that make it quite cheap computationally. However, the 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 [7], involves a Boolean check for intersection that only involves multiplication and division and does not involve expensive operations like square roots and 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 since 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, T_{abcd}, where *a*, *b*, *c*, and *d* are the nodes that define the tetrahedron and a_i, b_i, c_i, and d_i are the node coordinates. This signed volume is calculated using Equation 2.

$$6V(T_{abcd}) = \begin{vmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{vmatrix} = \begin{vmatrix} a_0 - d_0 & a_1 - d_1 & a_2 - d_2 \\ b_0 - d_0 & b_1 - d_1 & b_2 - d_2 \\ c_0 - d_0 & c_1 - d_1 & c_2 - d_2 \end{vmatrix} \quad \text{Eq. 2}$$

The result is six times the volume of the tetrahedron used to construct the equation. The sign of the volume, T_{abc}, is negative when the triangle formed by nodes *abc* forms a clockwise circuit when viewed from the observation point of node *d*. This Boolean test constitutes a topological primitive and is the fundamental building block for all TTI tests performed in this research.

Recall 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. To determine if two edges of each triangle cross the plane of the other, the following is done.

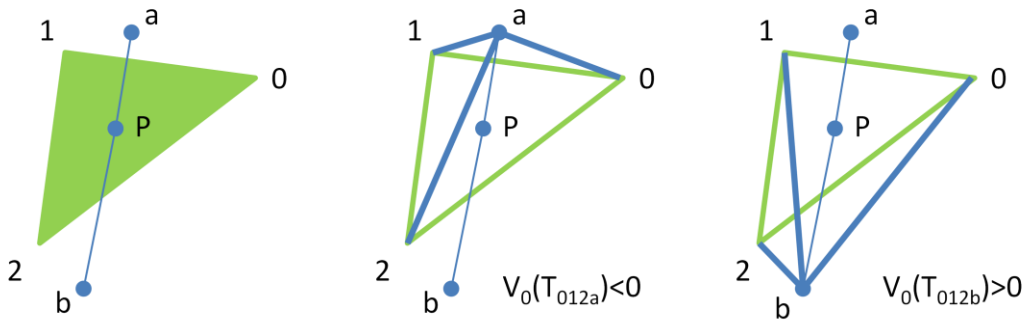


Figure 5.2 Triangle-Edge intersection test using topological primitive

Above in Figure 5.2, an example of using signed tetrahedral volumes to determine if a line segment pierces a plane is shown [7]. The signed volume defined by *(0,1,2,a)*, $V(T_{0,1,2,a})$, is compared to the signed volume *(0,1,2,b)*, $V(T_{0,1,2,b})$. If they are of opposite sign then the line segment pierces the plane of the triangle. This must be done for each edge to check to make sure that at least two edges from each triangle pass this check. Next, it must be determined if the line segment found to intersect the plane of the triangle pierces within the boundaries of the triangle.

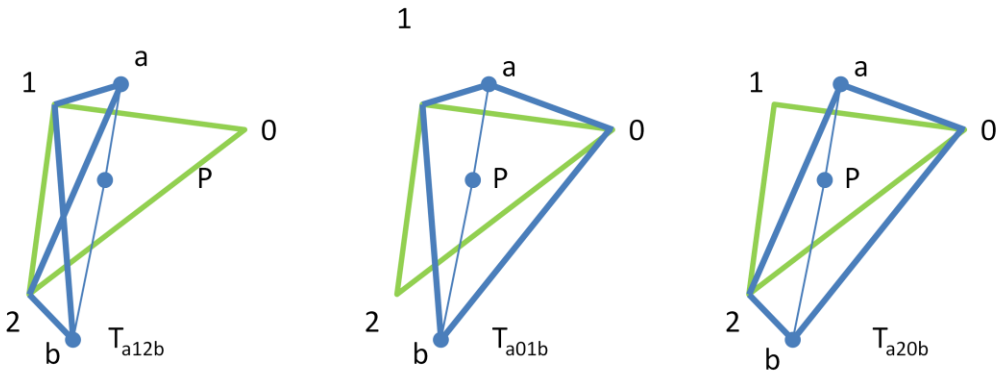


Figure 5.3 Triangle-Edge Intersection, Edge within boundary of Triangle

$$[V(T_{a,1,2,b}) < 0 \text{ and } V(T_{a,0,1,b}) < 0 \text{ and } V(T_{a,2,0,b}) < 0] \text{ or}$$
$$[V(T_{a,1,2,b}) > 0 \text{ and } V(T_{a,0,1,b}) > 0 \text{ and } V(T_{a,2,0,b}) > 0] \qquad \text{Eq. 3}$$

In Figure 5.3, an example of using signed tetrahedral volumes to determine if a line segment pierces a plane within the boundaries of a triangle is shown [7]. Three volumes must be checked to determine if the line segment pierces within the boundary of the triangle. The volumes are denoted $V(T_{a,1,2,b})$, $V(T_{a,0,1,b})$, $V(T_{a,2,0,b})$. If all of these volumes have the same sign, Equation 3, then the edge pierces within the boundaries of the triangle and the pair of triangles intersects.

A topological primitive is defined in [7] as, "an operation that tests an input and results in one of a constant number of cases." It is further stated that, "Such primitive can only classify, and constructed objects (like the actual locations of the pierce points…) 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 non-degenerate cases and zero represents some degeneracy involved with the geometry. By defining "zero" locally

for each pair of triangles tested for intersection, this tool becomes very robust and does not need computationally-expensive, exact-arithmetic routines. See the section on robustness later in this chapter for a more detailed explanation on how computational errors associated with degenerate geometries are handled.

Neighbor Tracing

Lo and Wang [8] 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 [8] proposed that once a pair of intersecting triangles was found that the topology of the mesh be used to construct the set of line segments defining the intersection. They denoted this process "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.

The determination of how to move through the mesh is determined on the type of intersection present. In Figure 5.4, three different types of intersections can be seen. Type 1 is a general intersection where one edge from each triangle intersects the other triangle. Type 2 is a special case of a general intersection where two edges from one triangle pierce the other triangle. Type 3 has only one edge that pierces. This means that of the two points that define the line segment that defines the intersection, one is a node in the existing geometry.

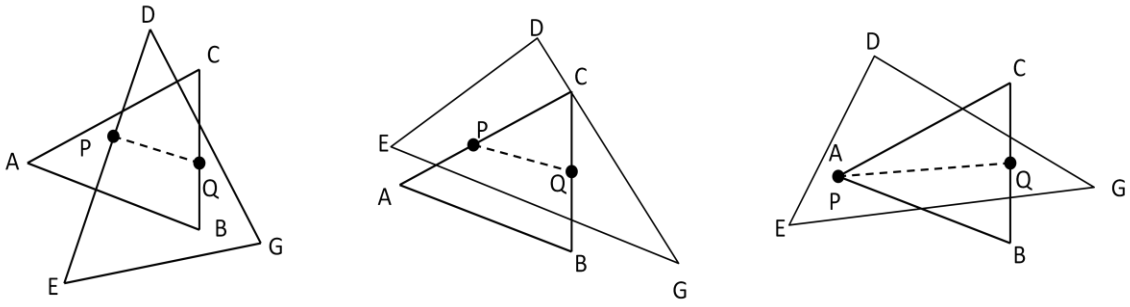


Figure 5.4 Three fundamental types of triangle intersections

In Figure 5.5 the three different types of intersections can be seen in place in a local mesh. This demonstrates how TNOIT can be used to construct the chain or loop of line segments that define an intersection. Starting with, as indicated in Figure 5.5a, triangles F1 and T1, if the intersection point, P, lies on the edge of F1, then the next pair to be tested for intersection should be T1 and F2—which is topologically adjacent to F1 across the edge. In Figure 5.5b, a similar process is used to move from the pair T1 and F1 to F1 and T2. However, in Figure 5.5c, the next intersection point is a node and therefore all of the topologically adjacent elements, T1-T5, must be tested for intersection with F1 before moving on.

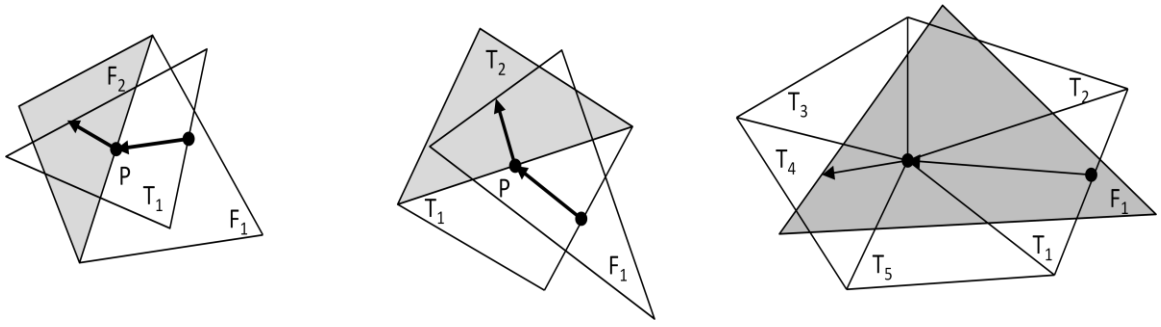


Figure 5.5 Three possibilities of how to move through a mesh using neighbor tracing

In addition to the above three intersection types, others which include degenerate geometries have been developed. As can be seen in Figure 5.6, an edge might not pierce within the boundaries of a triangle. If it does not, then it either 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 5.6a, an edge of T1 intersects an edge of F1. This means that both edges would have to be traversed in order to move to the next pair of intersecting triangles. In Figure 5.6b, 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, in order to move to the next pair of intersecting triangles.

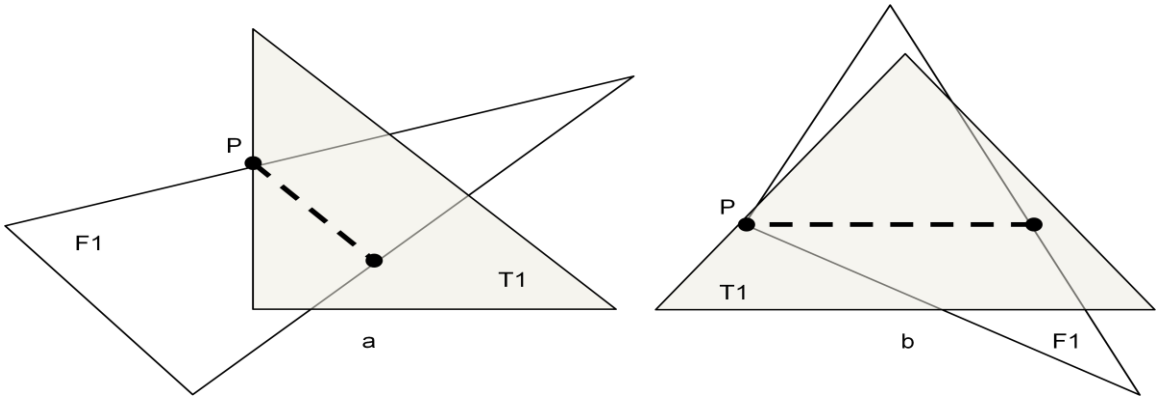


Figure 5.6 Degenerate possibilities of intersections

Local Repair

While tracing the segments through the mesh, the segments that lie in each triangle are stored for later use. These line segments represent the intersection between the two surfaces. In order to remove the intersection, the line segments must be inserted

into both surfaces. This would leave a set of non-manifold edges shared by the intersecting surfaces, but the intersection would be removed. 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 means that instead of a global set of edges to insert into the mesh, the problem can be broken into many smaller 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. This local, two-dimensional transformation is accomplished by rotating the geometry into the x-y plane using the equations given in the source code in APPENDIX A.3.

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 because 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 5.7a.

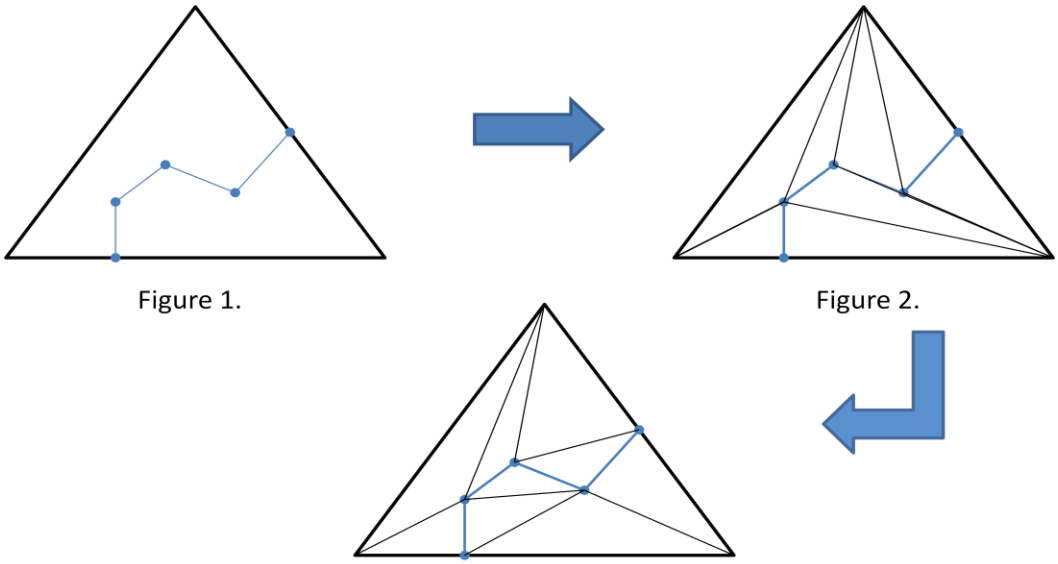


Figure 5.7 Local repair view edge insertion

The edges are inserted individually by first inserting the defining nodes and then recovering the edge, Figure 5.7b. A local min-max reconnection pass is then performed until no more edges fail the min-max test, Figure 5.7c. Each of these steps, node insertion, edge recovery, and local reconnection will now be discussed in more detail.

Node Insertion

In order to put edges into the triangulation, the nodes that define the edges must first be inserted. The process of finding the triangle that contains the node involves another topological primitive. Let the vector from node N_0 to node N_1, and node N_0 to node N_2 be denoted as follows in Equation 4a and Equation 4b.

$$\overline{N_{01}} = \left[N_{1x} - N_{0x}; N_{1y} - N_{0y}; N_{1z} - N_{0z}\right] \quad \text{a.}$$

$$\overline{N_{02}} = \left[N_{2x} - N_{0x}; N_{2y} - N_{0y}; N_{2z} - N_{0z} \right] \qquad \text{b.}$$

$$A_{0,1,2} = \frac{1}{2} \cdot \left(\left\| \overline{N_{01}} \otimes \overline{N_{02}} \right\| \right) \qquad \text{c}$$

Eq. 4

This topological primitive in this case is the area of a triangle. Equation 4b is used to calculate the area of a triangle. In two dimensions, the absolute value is removed because the only non-zero component will be the *z* component. The *z* component, along with its sign, is taken to be the area of the two dimensional triangle. The calculated area is positive if the nodes form a counter-clockwise circuit, i.e., the resulting vector is in the positive (+) *z* direction. In order to test if a node is within the boundaries of a triangle, three areas must be checked.

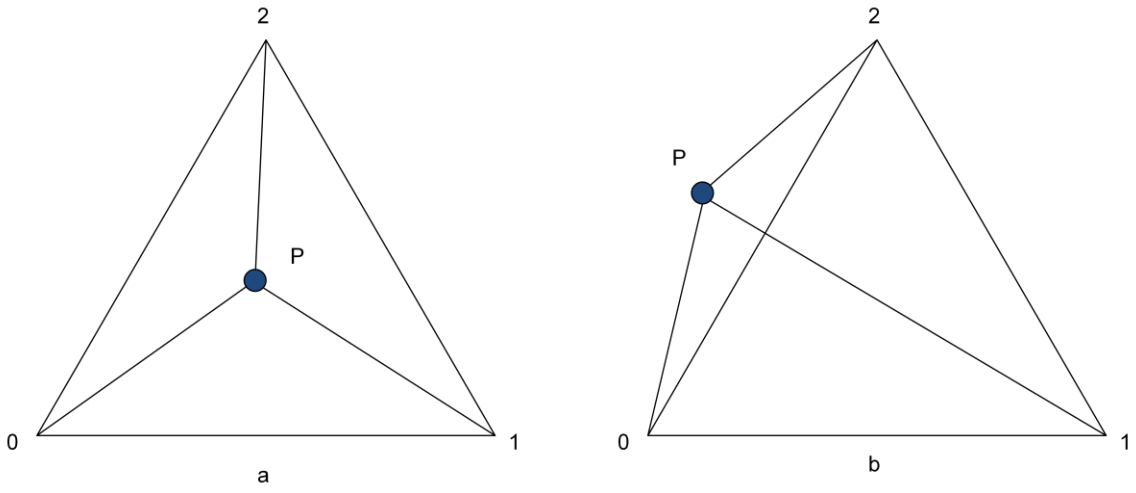


Figure 5.8 Containing triangle area check

In Figure 5.8a, the three areas that must be checked are the triangles formed by *(0,1,P)*, *(1,2,P)*, and *(2,0,P)*. If all of these areas are positive, the node, *P*, is in the

interior of the triangle, *(01,2)*. However, in Figure 5.8b, all three areas are not positive. The area of triangle *(2,0,P)* is negative. Therefore the edge, *(02)* is considered to be "associated with" the negative area. Because of negative area formed by *(2,0,P)*, the node *P* is not in the interior of triangle *(0,1,2)*. The two dimensional node-in-triangle check is used as a path finding mechanism for finding the containing triangle of a node. Consider Figure 5.9, in which the search for the containing triangle begins in the seed face. Since the node does not reside in the seed face, the edge that is associated with the negative area is traversed. For example, in Figure 5.8b, the searching algorithm would go to the element that is topologically adjacent to edge *(02)*.

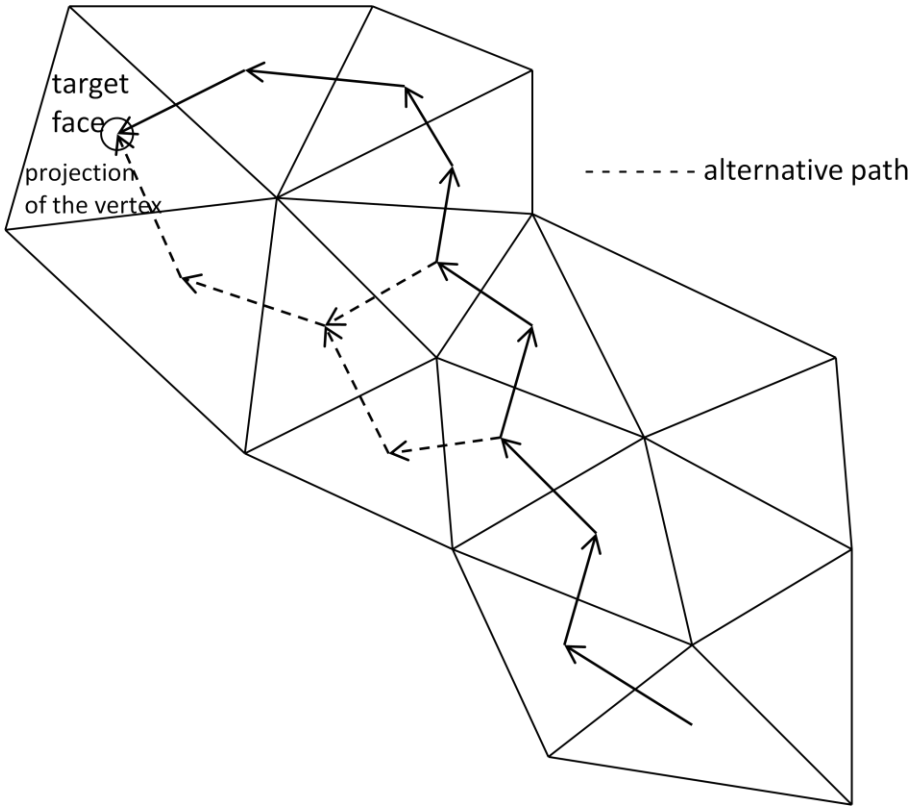


Figure 5.9 Two-dimensional, containing triangle search example

This process, calculating areas and traversing edges, is repeated until the containing triangle is found or the search fails because the node lies on an edge. If a containing triangle is found, it is split into three triangles, as seen in Figure 5.10a. If the node is found to lie on an edge, the edge is split as seen in Figure 5.10b.

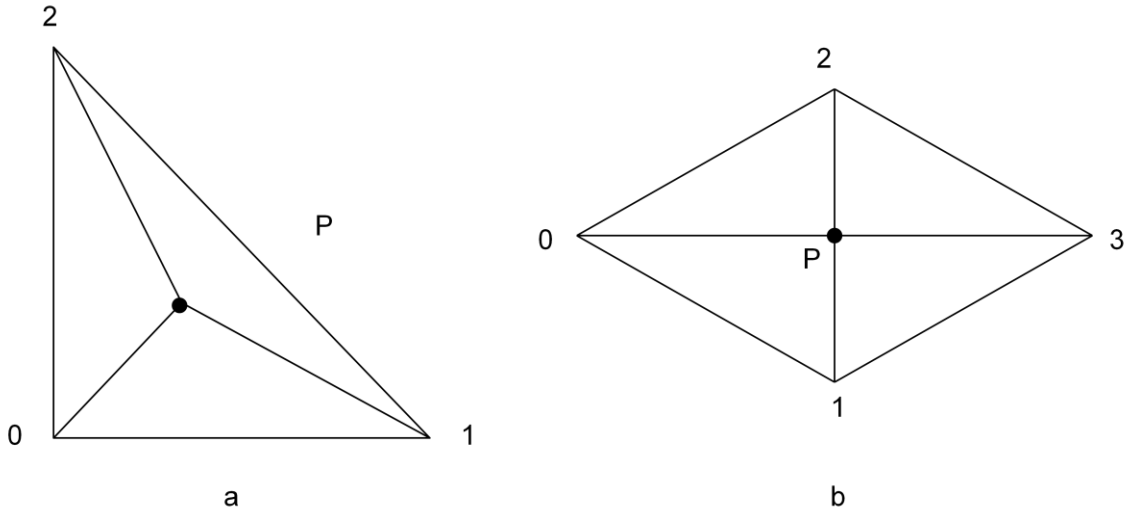


Figure 5.10 Triangle splitting and edge splitting example.

It is possible to not include the edge splitting option and rely on the 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.

Edge Recovery

Once the defining nodes of an edge are successfully inserted into the 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 (Figure 5.14) [24]. In order to recover the edge, a list of edges that should be swapped needs to be constructed. The equations used to determine if two edges intersect within some tolerance can be found in the source in APPENDIX A.5. It should be noted that the equations in APPENDIX A.5 do not calculate the point of intersection directly. They instead calculate the closest point on an edge to the other edge. The list of edges that should be swapped contains only the edges that intersect the to-be-recovered edge. An example of this can be seen in Figure 5.11. Starting at the node on the left, each edge that intersects the to-be-recovered edge is traversed—and stored—until the node on the right is found.

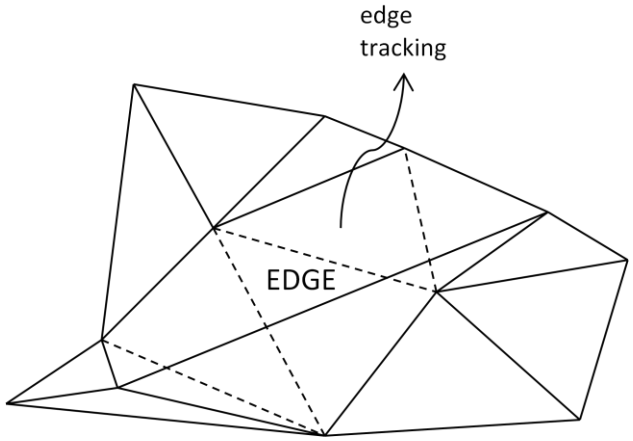


Figure 5.11 Finding edges that intersect the to-be-recovered edge using edge tracking

Once the list has been constructed, the following algorithm can be used to recover the desired edge [23].

1. Each edge of the set is swapped if
    a. *First Contraint*: its new swapped configuration does not create intersections
    b. *Second Contraint*: its new swapped configuration does not intersect the to-be-recovered edge.
2. If there are edge left unswapped in the list due to the constraints of 1(a) and 1(b) then the following strategy is performed.
    a. Relax the second constraint for the first unswapped edge and try to perform the swaps of the rest of unswapped edges. Flag the first relaxed edge still unswapped for the secong visit.
    b. A sweep of edges with both of the constraints being in effect is followed for swap.
    c. This trial scheme is continued until the edge is recovered or no swap could be performed due to geometrical validity, i.e. first constraint.
    d. If the edge is not recovered due to the first constraint then it is replaced with a set of edges forming a path between its end vertices. These edge pieces are recursively tried to be recovered.

Figure 5.12 Edge swapping algorithm

This process is demonstrated in Figure 5.13. The list of swappable edges includes edge, 1, 2, 3, and 4. Looping through the edges on Figure 5.13 the first pass we see that edges 1 and 2 can be swapped. Edge 3 cannot be swapped because it violates condition 1a from Figure 5.12. Once edge 4 is swapped, then edge 3 can be swapped in the second pass to arrive at the desired geometry. It is worth noting again that this process is guaranteed to converge to the desired result in two dimensions.

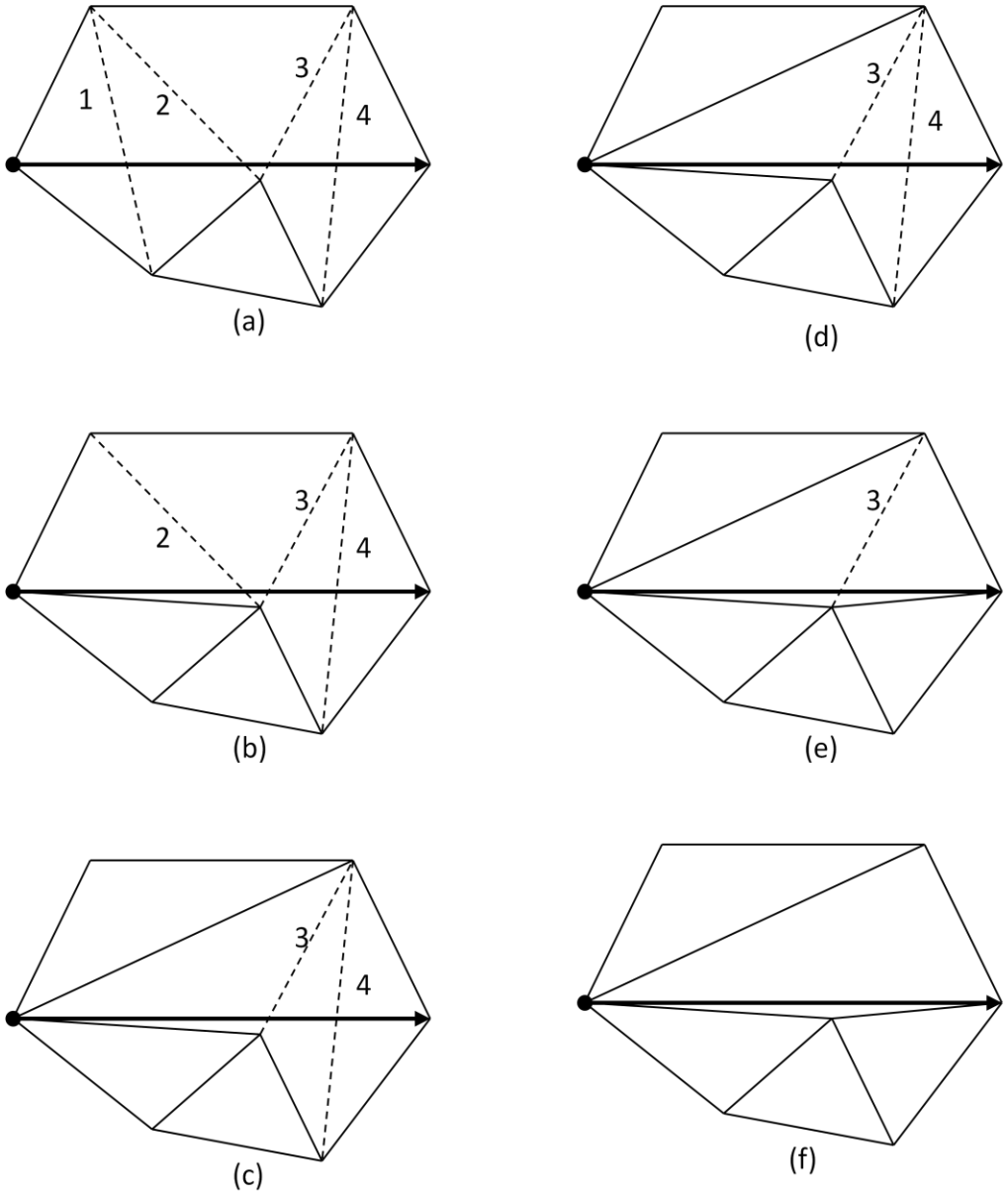


Figure 5.13 Edge recovery process via edge swapping

Local Reconnection

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 is the mesh. The aforementioned constraints are the inserted

edges. These edges must be present for the final geometry to repair the intersection. For a local edge to be reconnected, its reconnected state must reduce the maximum angle of the current state. An example of this can be seen in Figure 5.14.

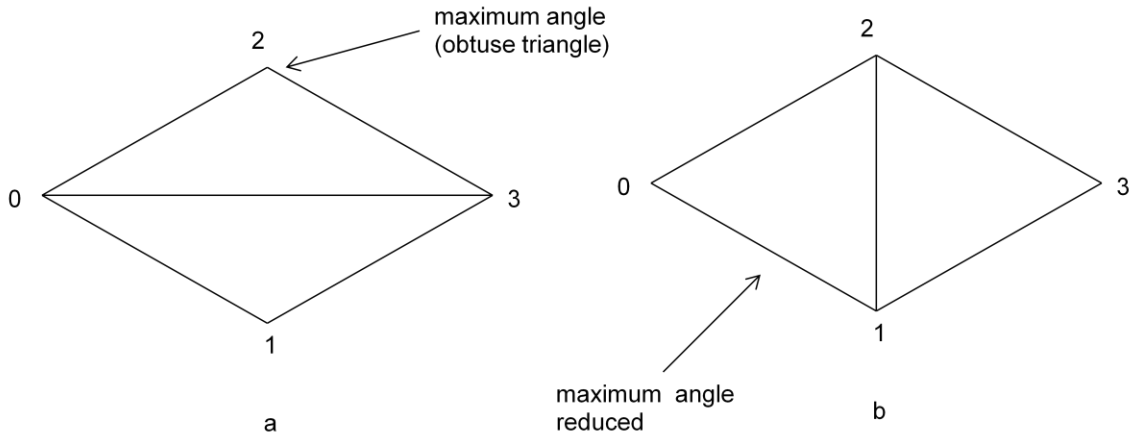


Figure 5.14 Local reconnection example using Min-Max criterion

In addition to the reconnection criterion, a stopping criterion was needed to ensure that the process did not reconnect geometry needlessly. If the maximum angle in the current or reconnected geometry is near ninety degrees, then the edge is left alone—since this can cause endless reconnections to be made while trying to improve the element quality. The loop that reconnects this geometry, since it avoids infinite reconnections, is guaranteed to converge [26].

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 update the topology of the three-dimensional mesh without any further transformations. This is accomplished through the use of "parent" nodes. The node class has a data member called a parent_GRX_NODE_ which is a pointer to a node. Since all of the geometry exists in three dimensions and then is transformed to two dimensions, each of the "two-dimensional" nodes has a "parent" from which it is derived or created. Creating the three-dimensional topology from the two dimensional, temporary mesh is as simple as creating all of the triangles that exists in the two dimensional mesh using the "parent" nodes instead of the "child" nodes for the connectivity. No additional calculations are used to transform the temporary mesh back to three dimensions—only the connectivity from the two dimensional mesh.

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 non-manifold meshes. The purpose of this tool is to aid in the production of watertight, manifold meshes. Therefore, some way of removing these non-manifold meshes needed to be developed, otherwise the intersection has removed one problem, intersecting geometry, and created another, non-manifold edges. One solution is to use the surface painting algorithm in described CHAPTER III with the EDGE_TEST() returning false for non-manifold, free-boundary, and surface-boundary edges. This post-processing step of surface painting would, if possible, "break-out" the surface defined in part or in whole by the non-manifold edges just created by the mesh intersection routines. These surfaces that have been "broken-out" can be removed or kept by the user based on the desired results.

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 to 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 that is locally degenerate may or may not be based on a global tolerance.

A degenerate volume will be treated differently depending on when it is encountered. For example, the TTI test consists of basically 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. If a degeneracy is encountered when testing if the edges of T_0 pierce within the boundaries of T_1, the edges might intersect. In the first case, a solution was sought to determine if the triangles intersected—to "break" the

degeneracy, or tie. In the second case, the edges are considered to intersect and this information is used to continue the TNOIT process.

The topological primitive used to determine if two triangles intersect will return one of three values, positive (+), negative (-), or zero (0). The zero-value threshold is the local tolerance—which is a fraction of the volume of the aforementioned bounding box. In the case of degenerate volumes for intersecting edges, the degeneracy offers useful information—the edges intersect. However, for determining if an edge pierces a plane, degeneracy offers no useful information. A technique called "Simulation of Simplicity" [7], [27] 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 since 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.

Since 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 is not applicable. The solution to this problem is to instead calculate the closest point on an edge. It does not matter on which edge the point resides since they will both be split with the same point. The addition of edge splitting produces higher quality elements along the line segments of intersection.

Neighbor Tracing

In an effort to make the neighbor tracing as accurate and robust as possible, the triangle pairs that are intersected are stored so that pairs are not intersected multiple

times. Edge-triangle pairs are kept as well as node-triangle pairs. These maps ensure that nodes and edges are not placed in triangles multiple times.

Edge Recovery

The edge recovery process involves two steps that need robustness, node insertion and edge swapping. Node insertion requires robustness so that the node is placed in the correct triangle or on the correct edge. This ensures that the geometry created is not self-intersecting. The containing triangle search takes place in two dimensions and an example of the check can be seen in Figure 5.8. A local tolerance similar to the one used for volume calculations is implemented here. In this case, a degenerate area means that the edge associated with the degenerate area should be split instead of the triangle. This effectively puts a buffer on each triangle that does not allow intersecting geometry to be created by placing a node in an incorrect triangle.

Edge swapping requires that two-dimensional line segments be tested for intersection. Instead of calculating whether edges exactly intersect, the closest points are calculated on each edge. The distance between these points is then compared to the longest of the potentially intersecting edges. If the distance falls below a set fraction of the "ideal" edge length, the edges are considered to intersect. This is another example of a tolerance being implemented locally instead of globally.

CHAPTER VI

ISOLATED BOUNDARIES

Introduction and Brief Overview

As discussed in CHAPTER II, current methods of mesh repair that are either node-pair based, or hole-filling algorithms are unable to repair isolated boundaries. Node projection is a process in which care must be taken to effectively and efficiently lessen or remove the effects of round-off and truncation error. Also, some method of reducing the algorithmic complexity of the node projection, specifically ray casting, from $O(n^2)$ must be developed. The repaired geometry must also be a valid mesh, free of degenerate and duplicate geometry. The following section is organized as follows: discussion and development of a node projection technique onto discrete surfaces, path finding and edge recovery, and gap filling and possible post-processing.

Edge Projection

Projection of nodes and NURBS curves onto NURBS surfaces is a standard operation in many CAD/CAE systems. However, to repair discrete geometry, a discrete analog to NURBS projection had to be developed. In order to repair the gap near the isolated boundaries, the edges had to be first projected onto the nearby discrete surface. This requires that the nodes defining the edges have to be projected onto the nearby discrete surface. In order to determine which direction to project the nodes, a projection direction had to be calculated. Then, a method of determining if that ray intersected any geometry was needed. The following sections describe each of these issues.

Node Projection Direction Calculation

In order to calculate a projection direction for each node, the surrounding geometry must be considered. Since the nodes being projected are boundary nodes, they necessarily have boundary edges topologically attached. These edges can each be said to have a normal direction that is perpendicular to both the edge and the normal vector of the attached element.

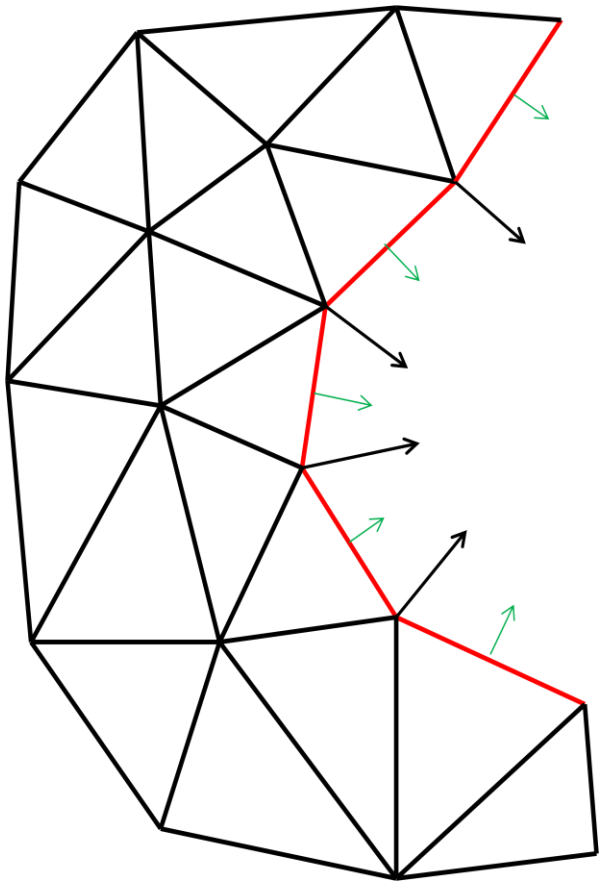


Figure 6.1 Free-boundary node-normal determination

A two-dimensional example of node and edge normals can be seen in Figure 6.1. The green normals attached to the edge are normals for the red boundary edges. The longer black normals attached to the nodes are the average of the topologically adjacent

boundary edge normals. Each boundary node has a unique normal associated with it. A least squares fit of all of the normals is used as a projection direction for the set of boundary nodes. The capability for each node to have its own projection direction exists, but the risk of intersecting normals is present. Projecting all of the nodes using the same projection direction guarantees that the geometry produced by filling the gap will not be self-intersecting.

Ray Casting

Ray casting is implemented here as a solution to the general problem of determining the first object intersected by a ray. This indirectly solves the problem of finding the closest discrete surface to a node in a defined direction. In CHAPTER IV, the octree query for determining the elements that potentially intersect the ray is discussed. Once a list of candidates for intersection is returned from the tree, the list is reduced to only those that the ray actually intersects. The element whose intersection point is closest to the node is then selected as the element into which the node is projected. The equations for determining if a ray intersects a triangle can be found in the source in APPENDIX A.2.

Node Projection

Once a projection direction is calculated, the node can be projected into the closest surface. The element the node is projected into is split into three small triangles. An example of this splitting can be seen in Figure 5.7. It is possible for the ray to intersect the discrete surface on an edge or a node. If a ray intersects an edge, then it is split at the intersection point—which is returned from the projection routine. If a ray intersects a node, then the existing node is returned from the projection routine. These

possibilities were incorporated so that nearly degenerate geometry was not created. Poor quality elements around the projected nodes could degrade the robustness of algorithms that rely on mesh quality.

Path Finding

Once the defining nodes of the to-be-projected edge are present in the projected-upon discrete surface, the edge must be recovered. This process is similar to the task discussed in CHAPTER V except here it is implemented in three dimensions. A direct three-dimensional implementation of the edge swapping algorithm presented in CHAPTER V cannot be implemented here because the to-be-recovered edge will not necessarily intersect any edge in the projected-upon surface. The task of finding edges that the projected edge would cross still exists but now another method of the "edge-tracking" procedure discussed above is needed. Karamete, et.al, [23] developed a method of finding a path of edges between the defining nodes of a to-be-recovered edge that is found by point-line projections and intersection checks.

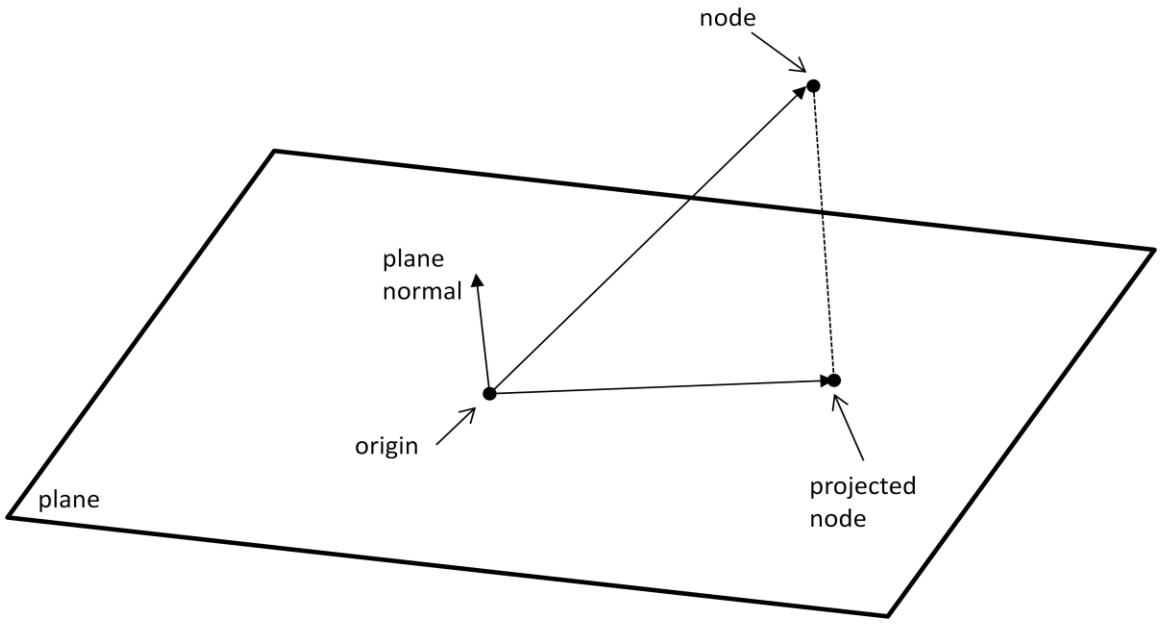


Figure 6.2 Projection of node into plane

Given the geometry shown in Figure 6.2 of a plane defined by an origin, *origin*, and a normal, *plane_normal*, find the projection of the vector pointing from *origin* to *node*. This projected vector will be defined by *origin* and the *projected_node*. In Equation 5 the equations used to accomplish this task are given. The *plane_normal* must be normalized before this calculation is done. The vector, *vec*, is projected onto *plane_normal* via a dot product. With this information, the coordinates of *projected_node* can be found by traveling from *node* in the direction of *plane_normal* a distance of *vec_dot*. Note, *vec_dot* is a negative value since the projection is towards the plane, *plane*.

$$\overline{vec} = \overline{node} - \overline{origin} \quad \text{a.}$$

$$vec\_dot = \overline{vec} \bullet \overline{plane\_normal} \quad \text{b.}$$

$$if\left(vec\_dot > 0\right) \rightarrow vec\_dot = -vec\_dot \quad \text{c.}$$

$$\overline{projected\_node} = \overline{node} + \overline{plane\_normal} \cdot vec\_dot \quad \text{d.}$$

Eq. 5

For each point-line projection seen in Figure 6.3, the origin of the plane is defined at the last point of intersection calculated on the edges in the projected-upon mesh, and the to-be-recovered edge. The *plane_normal* is the normal of the current triangle.

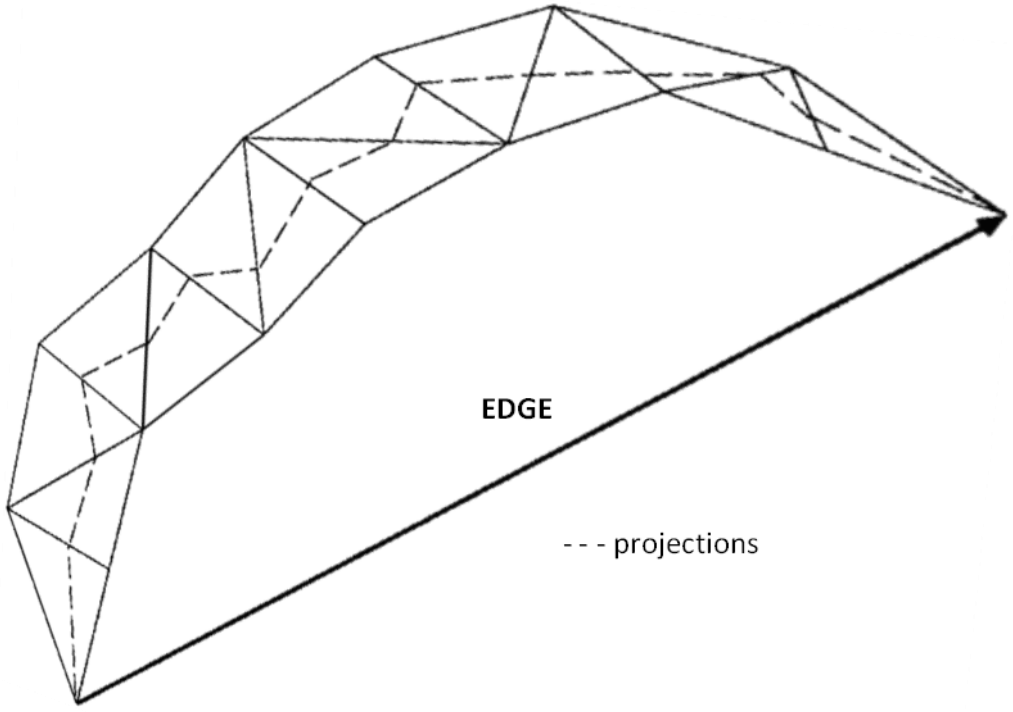


Figure 6.3 Limited point-line projections

Figure 6.3 is an example of the use of controlled point-line projections to find the path of edges between nodes. In order to accomplish this, the edge is projected into the current triangle and then that projected edge is checked for intersection with the edges of

the triangle. The newly intersected is edge is then traversed to arrive in a new triangle. The edge is projected into the plane of this triangle and the process is repeated until the end node is found. The result is a list of edges that would intersect the to-be-projected edge.

Whole Edge Recovery vs. Edge Recovery with Edge Splitting

Once a list of edges that the to-be-projected edge would intersect is constructed, the issue of how to recover the edge is raised. The algorithm in Figure 5.12 is the solution of choice in [23]. However, this can lead to highly skewed results—similar to the results seen in Figure 6.4.

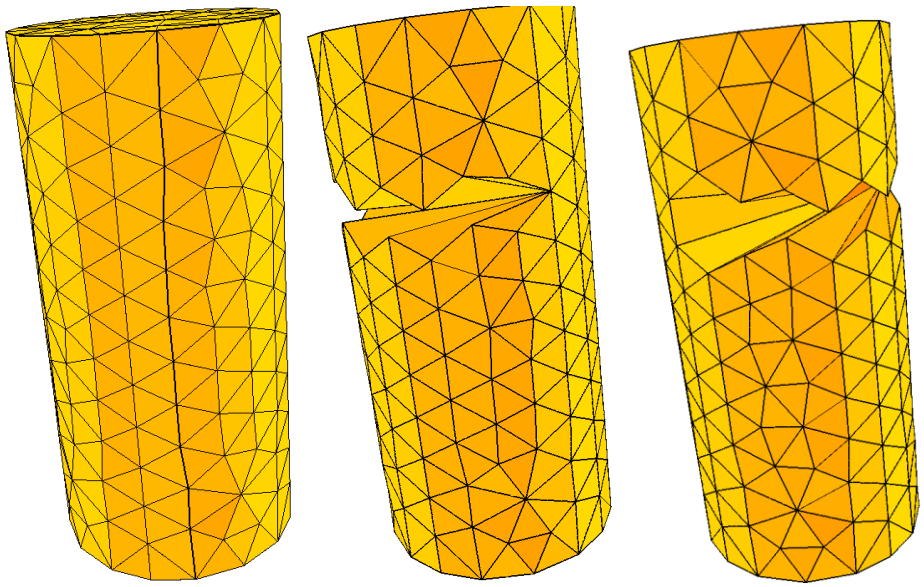


Figure 6.4 Skewed geometry resulting from 3D edge recover algorithm.

The edge that was recovered in Figure 6.4 traversed a relatively large number of edges and a relatively large amount of curvature. The result of recovering the whole edge is a distorted geometry that destroyed the original shape of the model. This can be seen in the

center and right cylinders in Figure 6.4—which are from the right and left sides of the cylinder respectively.

Instead of recovering the whole edge, the proposed solution was to split the to-be-recovered edge into as many pieces as the number of entries in the list of swappable edges. Also, each edge that is in the list of swappable edges is split where the to-be-recovered edge intersected while projected into the plane of a triangle. This process would not recover the whole edge, but it would retain the curvature of the projected-upon surface. A two dimensional example of this can be seen in Figure 6.5.

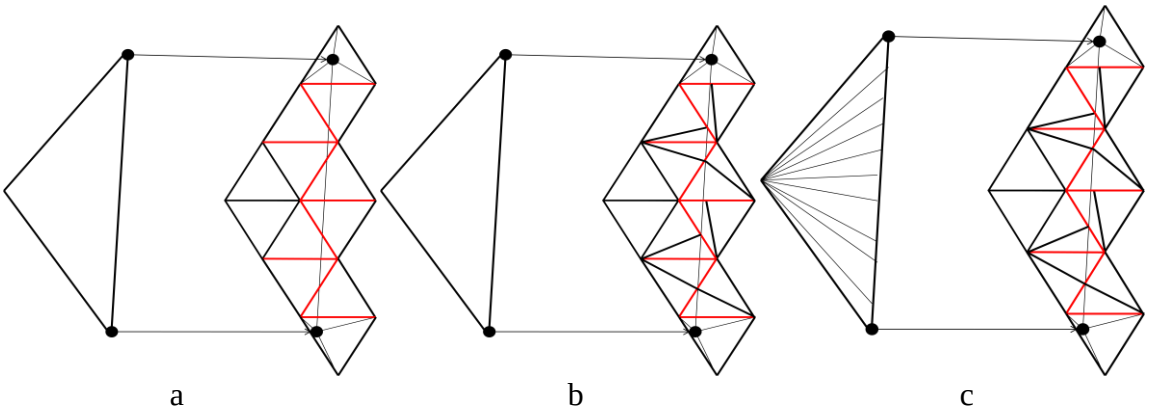


Figure 6.5 Edges split instead of swapped

The red edges in Figure 6.5a are the edges that intersect the to-be-recovered edge. They have been split instead of swapped to arrive at the results shown in Figure 6.5b. The projected edge is also split as can be seen in Figure 6.5c. The purpose of splitting the edges on the projected-upon surface is to retain the curvature and prevent the unintentional creation of distorted or poor quality triangles. The purpose of splitting the original edge is to arrive at a state where all of the points created in the projected-upon

surface have a pair on the original edge. This makes filling the gap near the isolated boundary a straightforward process. It should be noted that the original edge can still be recovered after all of the splits have been performed, if desired, by simply gluing the interior nodes together on the projected-upon surface and the original edge.

Filling the Gap

After the node projection, and subsequent edge splitting, the geometry resembles Figure 6.5c. The only thing left to do in order to repair the isolated boundary is fill the gap. If only the set of edges that comprise the projected edge and the original edge is considered, then the problem becomes a mesh generation problem with the boundaries defined by the aforementioned edges, Figure 6.6a.

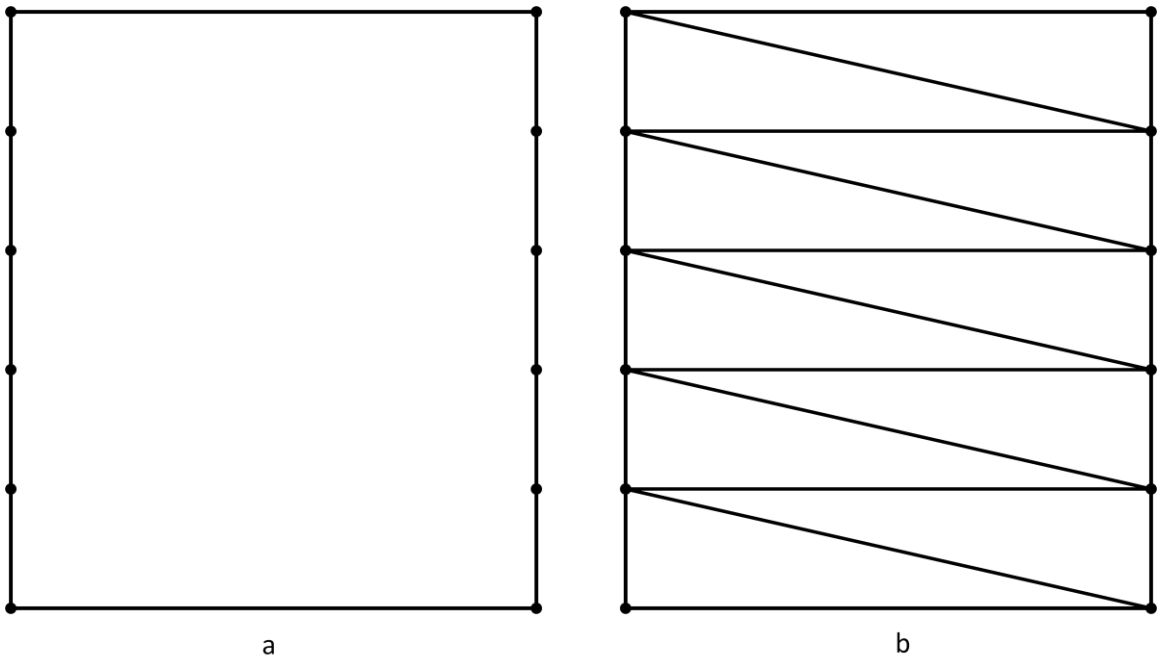


Figure 6.6 Filling Gap Defined by Isolated Boundary and Surface

There is no need to place any nodes in the interior of the gap since a triangulation that fills the gap is all that is required. Therefore, since the nodes on the surface all form a pair with a node on the original edge, filling the gap is straightforward (Figure 6.6b).

Post Processing

Now that the gap has been filled with new triangles, a post-processing step can be added. If the distance between the nodes on the original edge and the projected edge fall below a given tolerance, the edge that is defined by those two nodes can be collapsed. This step is not necessary to repair the gap since it is already filled at this stage. However, it could be useful if the desired result was actually the edges glued to the discrete surface instead of merely projected.

The process of inserting the line segments, or edges, defining the projection into all of the appropriate discrete surfaces necessarily creates non-manifold meshes. The purpose of the isolated-boundary repair tool is to aid in the production of watertight, manifold meshes. Therefore, some way of removing these non-manifold edges needed to be developed, otherwise the intersection has removed one problem, isolated boundaries, and created another, non-manifold edges. One solution is to use the surface painting algorithm defined in CHAPTER III with the EDGE_TEST() returning false for non-manifold, free-boundary, and surface-boundary edges. This post-processing step of surface painting would, if possible, "break-out" the surface defined in part or in whole by the non-manifold edges just created by the edge extension. These surfaces that have been "broken-out" can be removed or kept by the user based on the desired results.

Robustness

Node Projection/Ray Casting

In an attempt to increase the robustness of this tool, the ability for a projected ray to intersect a node, edge or triangle was added. This improves mesh element quality around the projected node. It also ensures that the path finding algorithm, which relies on controlled point-line projections, is not hindered by the node projection process due to poor quality elements. In the ray-element intersection routine seen in APPENDIX A.2, the return value is non-zero if the edge intersects the triangle. The parameters returned are *t*, *u*, and *v*. The first parameter, *t*, is the length along the normalized direction vector, *dir*, along which the intersection point lies. The next two parameters, *u* and *v*, are the coordinates of the intersection point in the local *u*-*v* space of the triangle defined by *vert0*, *vert1*, and *vert2*. An example of the *u*-*v* space formed by a triangle can be seen in Figure 6.7.

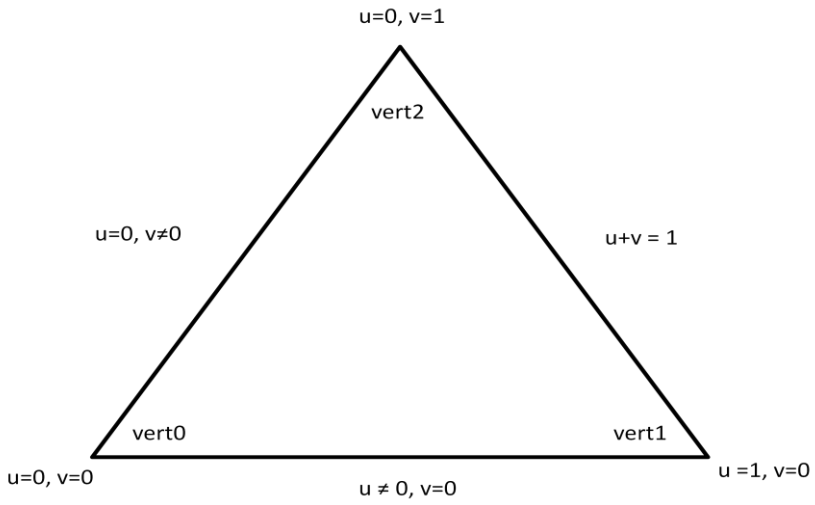


Figure 6.7 Triangle in local *u*-*v* space

The values that are returned for *u* and *v* define where on the triangle the ray intersects. Given a tolerance, *tol*, if $u<tol$ or $v<tol$ that would indicate that the ray intersects an edge or a node. If the last statement were true, then the following conditions would indicate that the ray intersects a node:

$$\text{if}\{(u < tol \text{ and } 1.0 - v < tol) \text{ or } (v < tol \text{ and } 1.0 - u < tol) \text{ or } (u + v < tol)\}$$

If the above check fails to indicate that the ray intersects the node, then the ray intersects an edge. Using the *u*-*v* map returned from the ray-triangle intersection test, a robust method of determining if the ray intersects a triangle on the interior or on an exterior entity was developed. In addition, since the values of *u*, *v*, and *t* are on the order of 1, some multiple of round-off error can be easily implemented as a tolerance—this is was implemented here.

Path Finding

Path finding between nodes is an integral part of the isolated boundary repair tool. The original edge cannot be projected onto a surface if a path between the projections of the defining nodes cannot be found. Because the edge line is *projected* into the triangle's plane as opposed to *rotated*, the projected edge line might not intersect any of the edges of the triangle. This could happen, for instance, if the edge is nearly perpendicular to the plane of the intersecting triangle. One specific instance of this would be that the points defining the projected edge are on nearly opposite sides of a cylinder (Figure 6.4). A solution to this problem is to extend the line out to infinity before projection to make sure that the projected edge line intersects at least one of the intersecting triangle's edges.

However, the solution implemented here is to first project the line onto the plane and then to extend it at least three times the length of the longest edge of the intersecting triangle. Again, a local tolerance is felt to be superior to a global tolerance. Also, since this tool might be implemented on different hardware, a hard-coded number representing infinity could prove problematic.

Another issue that was addressed is that during the edge-line projection step of the path-finding algorithm the edge line, once projected into the plane of the triangle, might be parallel to an edge and therefore does not properly intersect any of the edges. If a parallel edge is found, then the other node defining the edge is used as the next point in the process of finding edges that intersect the to-be-recovered edge. However, in this case all of the elements topologically attached to the node have to be searched for edge intersection. If instead of using the opposite node of a parallel edge an intersection point were calculated on an edge, the resulting point would be coincident to one of the end points of the parallel edge. If this edge were then split as usual, degenerate geometry would be created. Therefore, the ability to use an opposite node of an edge instead of calculating an intersection point with a nearly parallel edge prevents the unintentional creation of degenerate geometry.

The final issue addressed is the possibility of creating intersecting geometry. The edge-projection process begins with node projections. Nodes will be projected into the closest element along a defined vector. A path between these nodes will be found and then the edge can be recovered. However, if the path between the nodes travels through a part of the mesh that the original edge cannot see then the edge cannot be projected and the algorithm moves on to the next edge in the list. This concept is demonstrated below in Figure 6.8. On the left is an example geometry where the edge can see the entire

projection path between vertices. In Figure 6.8, right the edge cannot see the entire projection path using the projection vector (blue arrows) and therefore cannot be projected onto the surface. For if it were projected onto the surface the geometry generated to fill the gap would intersected the projected-up surface.

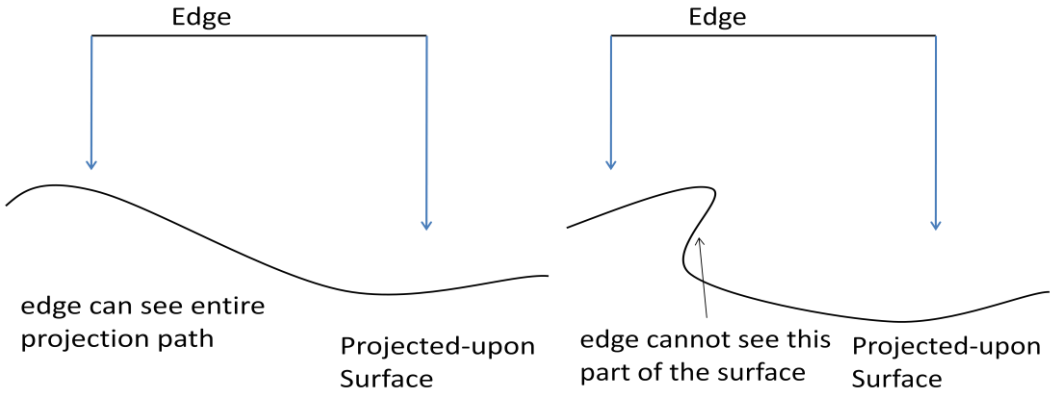


Figure 6.8 Edge Projection Failure Condition Comparison

CHAPTER VII

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 laptop with a 2.33GHz Intel® Core™2 CPU with 4.00 GB of RAM. The operating system used was 64-bit, Ubuntu Linux 9.10.

Intersecting Mesh

The following examples show various results from different size meshes.

Nearly Parallel Geometry

The following example demonstrates the robustness of triangle intersection routines. The geometry seen in Figure 7.1 is nearly parallel. The pink surface is one degree rotated from the flat plane that is defined by the purple triangle. A deviation of less than one degree could not be visually demonstrated well enough to make an effective example. The two geometries present in Figure 7.1 can be seen individually in

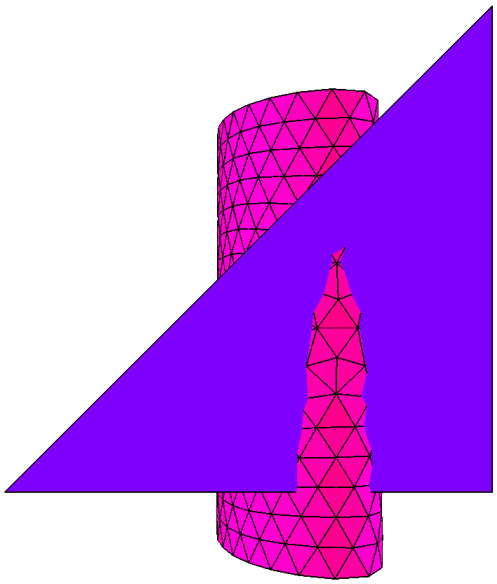


Figure 7.1 Intersection of Nearly parallel Geometry

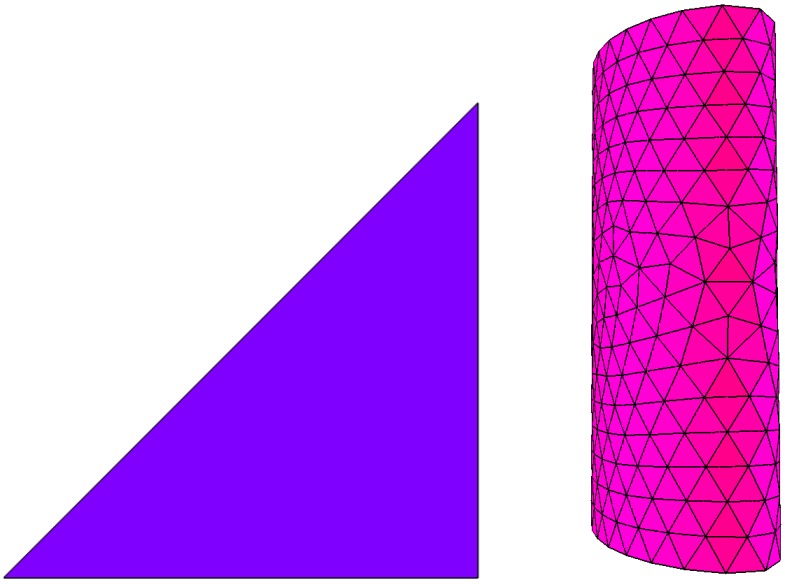


Figure 7.2 Individual Surfaces of Nearly Parallel Geometry

In Figure 7.3, Figure 7.4, and Figure 7.5 the nearly parallel geometry can be seen after the intersection process. The lines of intersection form a closed loop on the triangle

and that surface has subsequently been separated into two distinct, topologically adjacent surfaces. These can be seen in Figure 7.4, left. A close-up of the lines of intersection inserted into the curved surface can be seen in Figure 7.4, right. In Figure 7.5, a close-up of the small gap formed by the one degree rotation of the curved surface can be seen. This figure best demonstrates how close these surfaces are to being parallel.

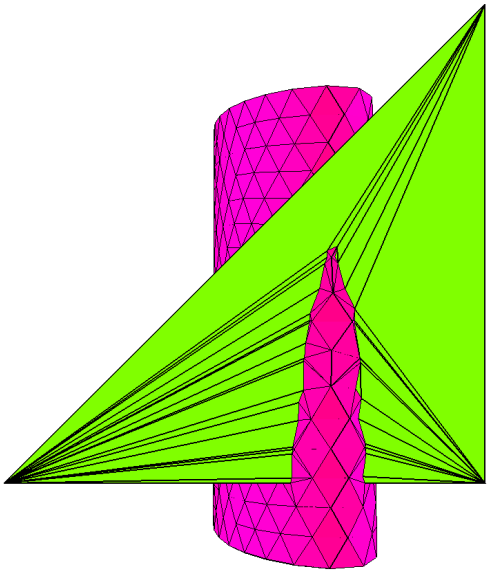


Figure 7.3 Intersection Nearly Parallel Geometry, Repaired

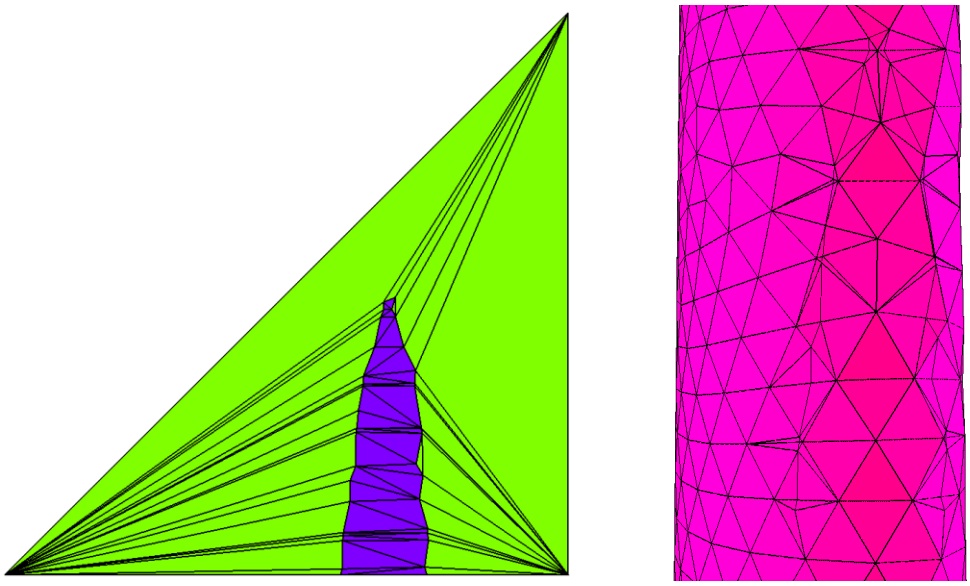


Figure 7.4 Isolated Surfaces of Nearly Parallel Geometry, Repaired

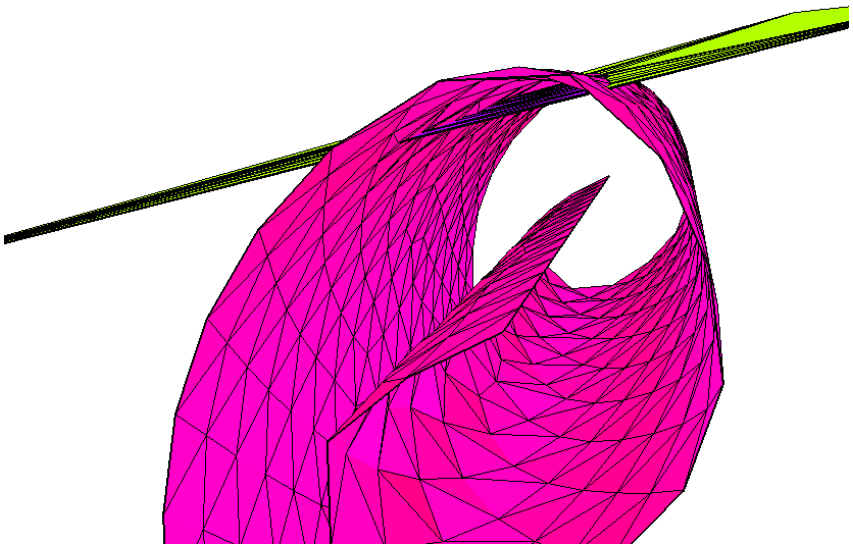
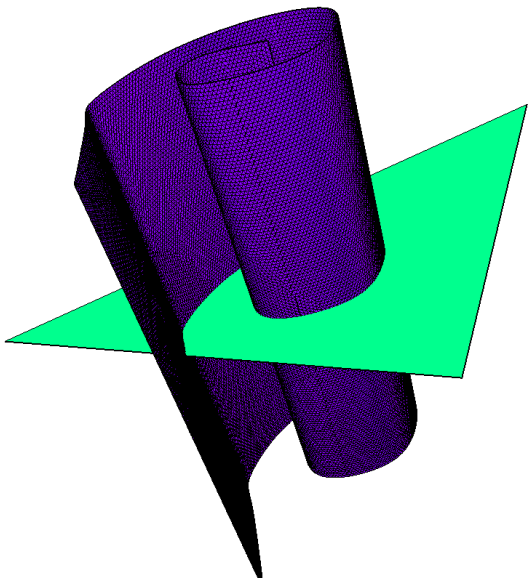


Figure 7.5 End-on view of Intersection of Nearly Parallel Geometry, Repaired

Many Edges in One Triangle—Simple

The following example demonstrates the local repair functionality discussed in CHAPTER V. Figure 7.6 shows the finer mesh intersecting the large triangle. 424 edges were inserted into the large triangle by intersecting it with a finer mesh. The results from the intersection create 720 triangles from the large triangle and can be seen in Figure 7.7. Cases such as this demonstrate the robustness of the edge insertion and recovery algorithm.



s

Figure 7.6 Example of inserting many edges into one triangle

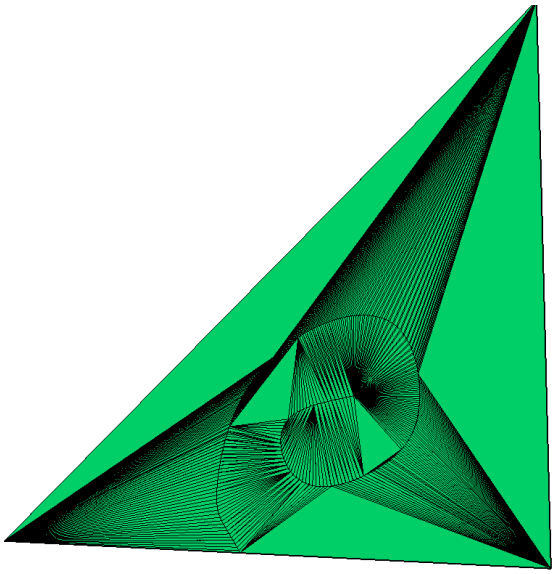


Figure 7.7 Results from inserting many edges into one triangle

Many Edge in One Triangle—Complex

In this example, the purpose is to show a real-world test case that demonstrates the robustness of the intersection tool through intersecting many, smaller triangles with two, larger triangles. Figure 7.8 shows the rear under-carriage of an SUV [29] where it intersects the under-body plate. This model was created for graphical purposes and only exists in discrete form. In order to repair the intersections and create a watertight geometry the discrete surfaces must be intersected. In addition there is no possibility of improving the quality of the elements defining the intersection.

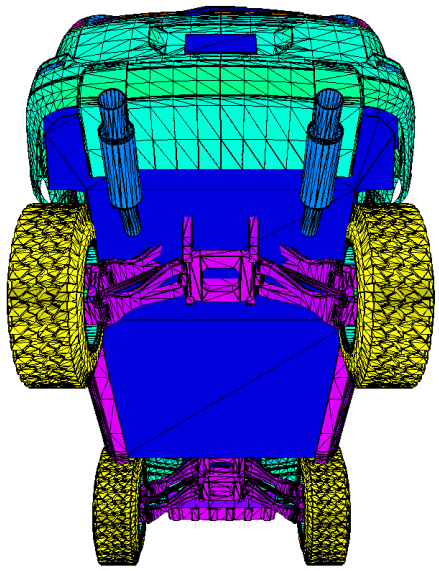


Figure 7.8 SUV Suspension, Intersection Example

In Figure 7.9, a close-up view of the intersecting geometry can be seen. The suspension assembly (purple surface, Figure 7.10) intersects the under-body plate (blue surface, Figure 7.11) with ten individual components. These figures also show that the plate, since it is flat, is defined by very few triangles.

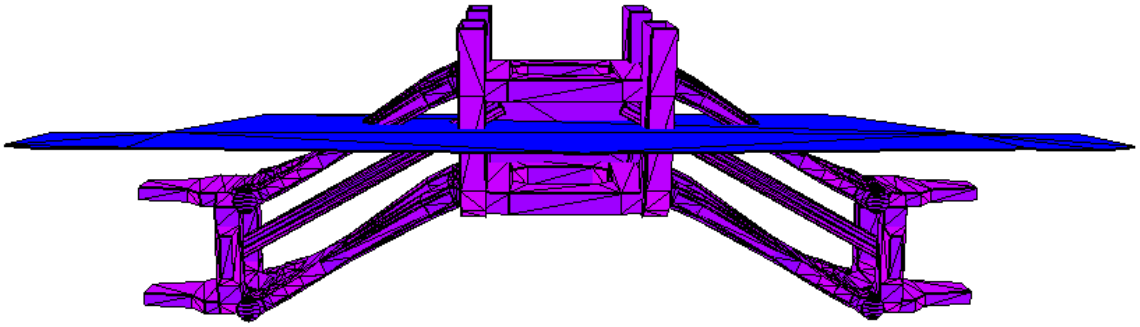


Figure 7.9 Close up View of Intersecting SUV Suspension and Under-body Plate

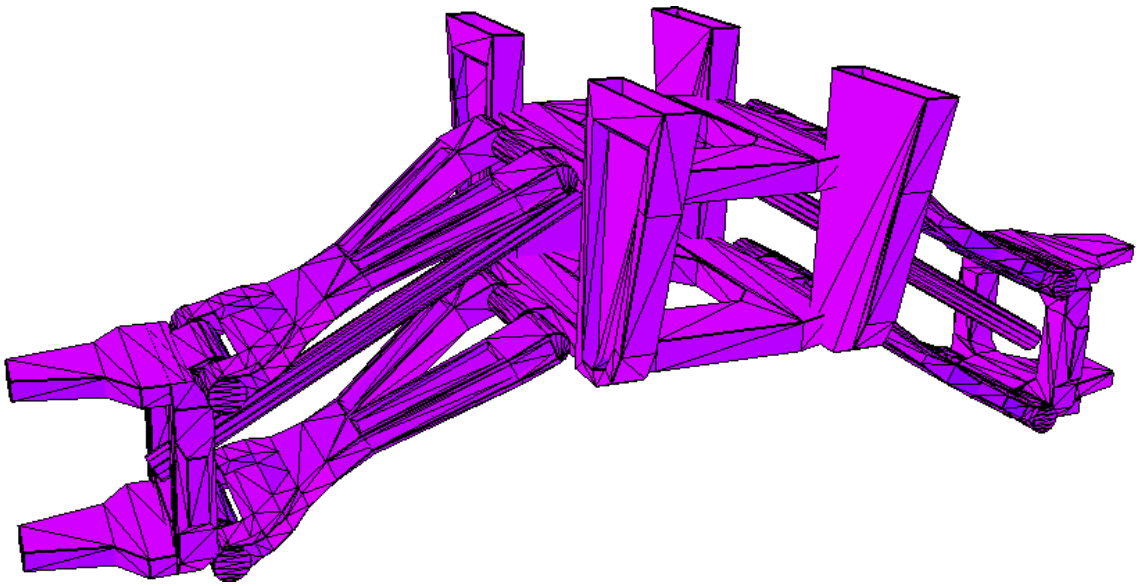


Figure 7.10 SUV Suspension Assembly

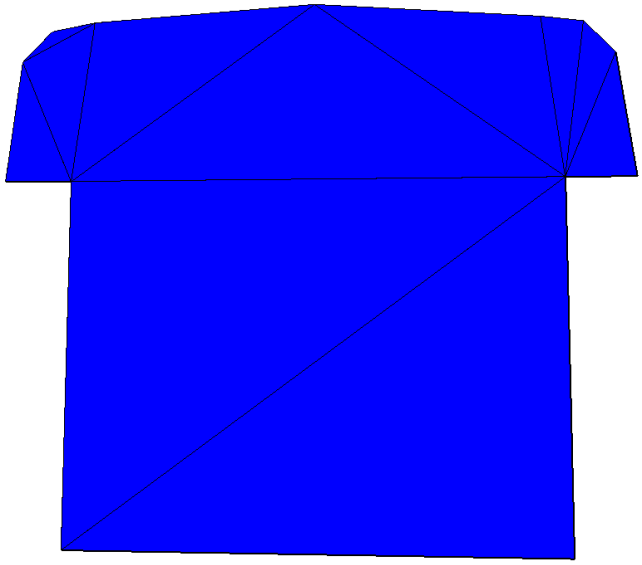


Figure 7.11 SUV Under-body Plate

In Figure 7.12 and Figure 7.13 the under-body plate of the SUV after intersection can be seen. The plate has been intersected with the ten individual components of the suspension assembly to arrive at this result. This figure shows that the two triangles that formed the under-body plate (Figure 7.11) have been split into 318 triangles to complete the intersection. The multi-colored shapes in the middle of the under-body are distinct surfaces formed by the edges that define the intersections and were broken out using the surface painting algorithm discussed in CHAPTER III.

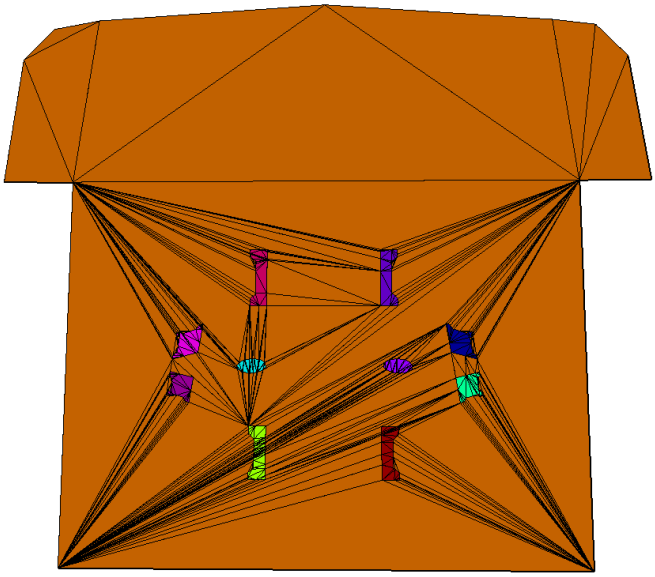


Figure 7.12 SUV Under-body Plate After Intersection

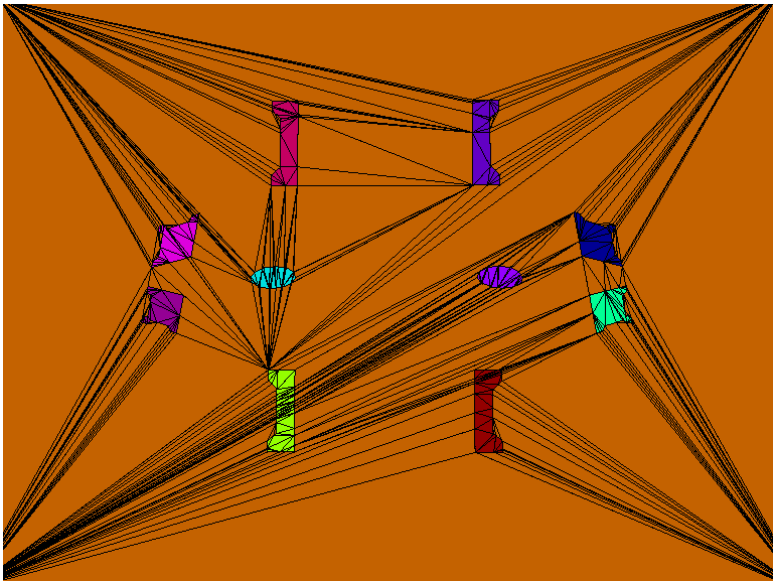


Figure 7.13 Close-up of SUV Under-body Plate After Intersection

In Figure 7.14 and Figure 7.15 the SUV suspension assembly can be seen after intersection. The multi-colored surfaces (except for purple) were broken out of the

assembly using the aforementioned surface-painting algorithm. The line of intersection with the flat plate can easily be seen in the following figures and is consistent and accurate for each component.

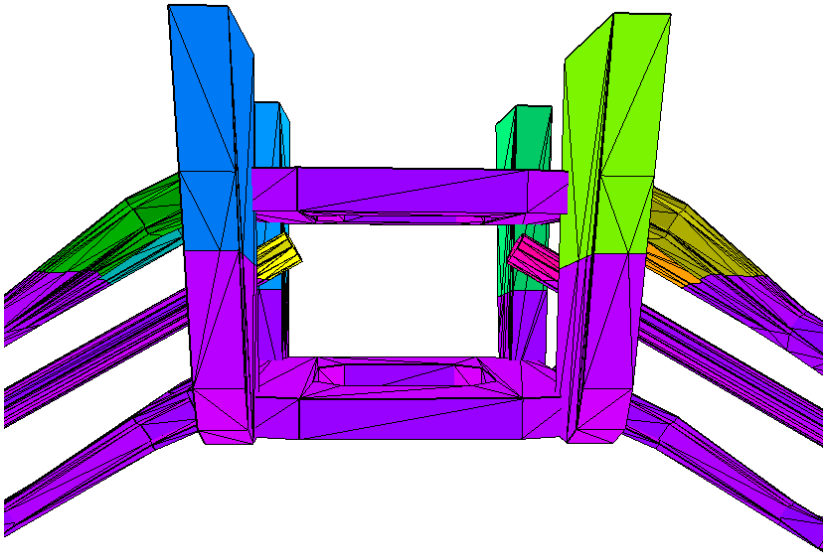


Figure 7.14 Close-up of SUV Suspension Assembly After Intersection, Head-on

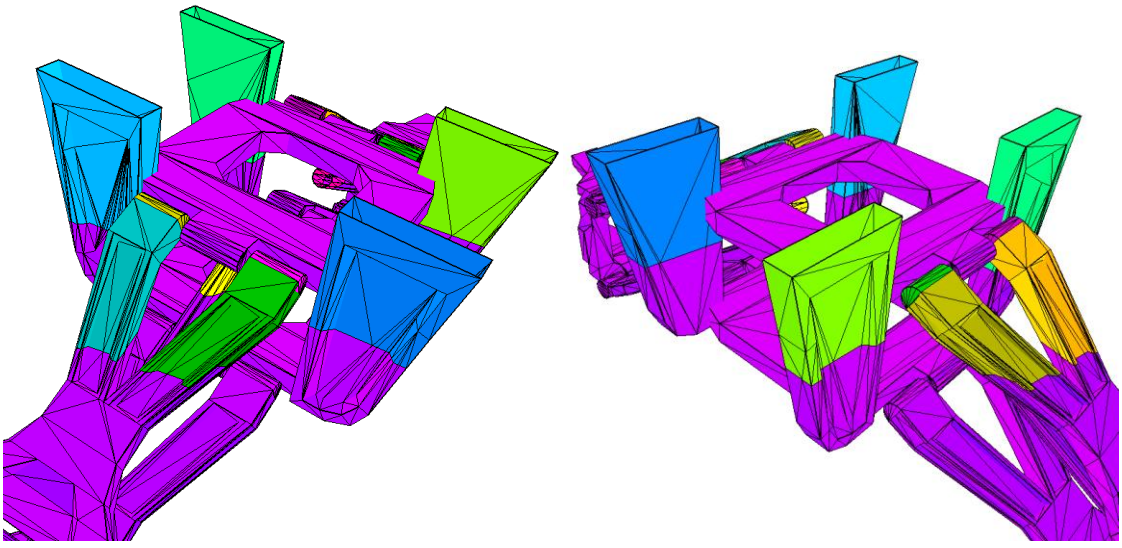


Figure 7.15 Close-up of SUV Suspension Assembly After Intersection, Isometric Left (left) and Isometric Right (right)

In Figure 7.16 and Figure 7.17 the result of removing the surfaces that were created during the surface painting can be seen. These figures show the consistent intersection formed with the flat plate. Figure 7.18 shows a close-up of the underside of the under-body plate where the suspension assembly intersected it. In this figure, the large number of triangles created during the intersection can be seen. After the intersection, surface painting, and removal of the interior/undesired geometry the suspension assembly is topologically adjacent to the under-body plate and no free-boundary edges were created.

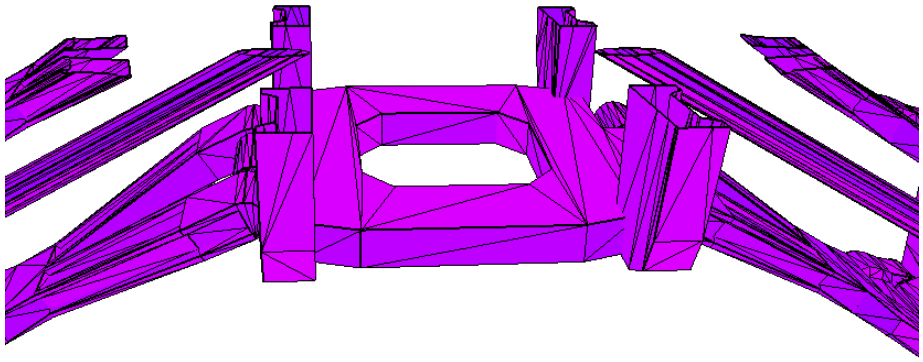


Figure 7.16 Close-up of SUV Suspension Assembly After Intersection with Interior/Undesired Geometry Removed

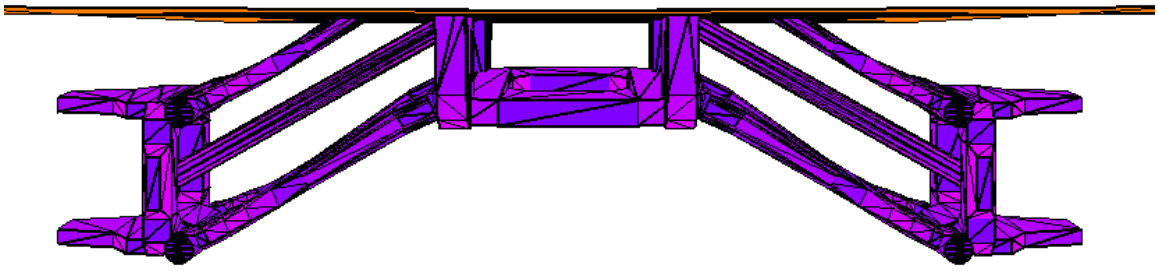


Figure 7.17 SUV Suspension Assembly and Under-body Plate After Intersection

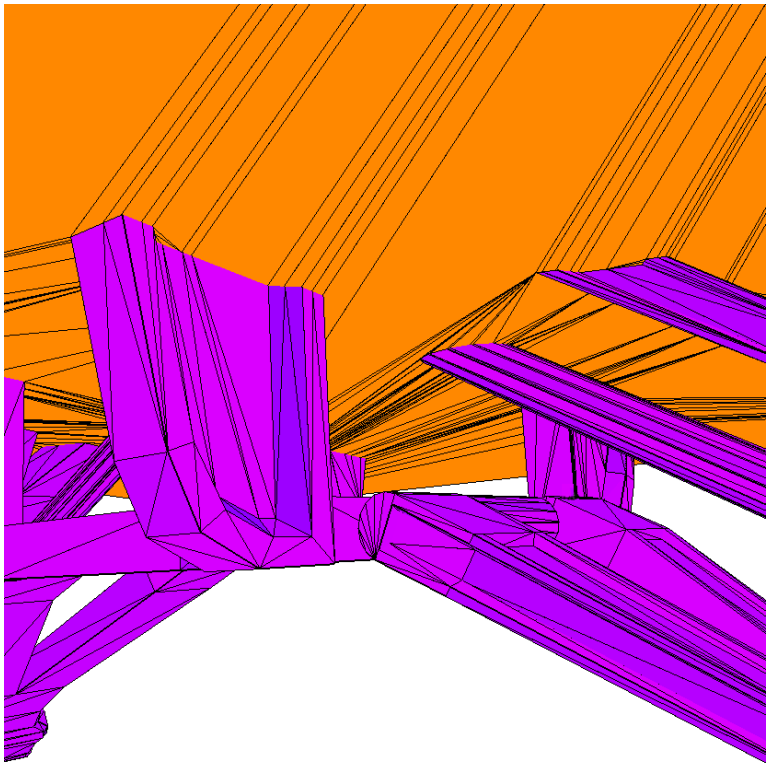


Figure 7.18 Close-up of SUV Suspension Assembly and Under-body Plate After Intersection

Loop Formed

The following example demonstrates a less severe example of inserting many edges into one triangle. However, the feature of interest in this example is the formation of a closed loop through the intersection process. In Figure 7.19, a cylindrical mesh is shown to intersect a coarse, triangular mesh. The closed loop formed by intersecting the surfaces, as seen in Figure 7.20 and Figure 7.21, is not defined by nodes that existed in the original geometry. The loop is formed completely by nodes that were calculated.

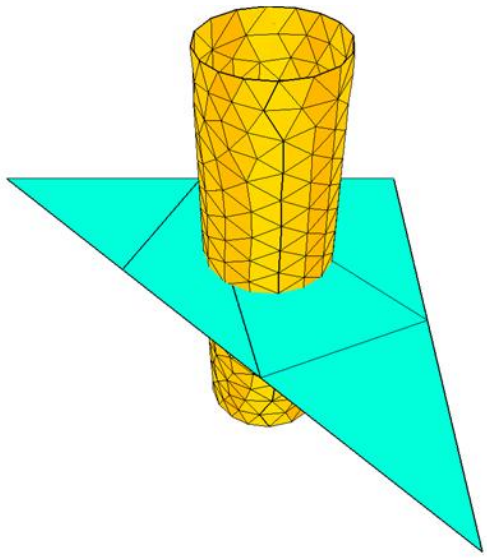


Figure 7.19 Cylinder Intersecting Coarse Triangular Mesh

In Figure 7.20 the coarse triangular mesh is shown after intersection. The lines of intersection have formed a closed loop in the surface. This allowed the surface-painting algorithm discussed in CHAPTER III to "break out" the surface bounded by the lines of intersection (purple). The lines of intersection also formed a closed loop in the cylindrical mesh (Figure 7.21 and Figure 7.22). The surface-painting algorithm was used

to separate the cylindrical surface into two surfaces, yellow and pink, bounded by free boundaries and the lines of intersection.

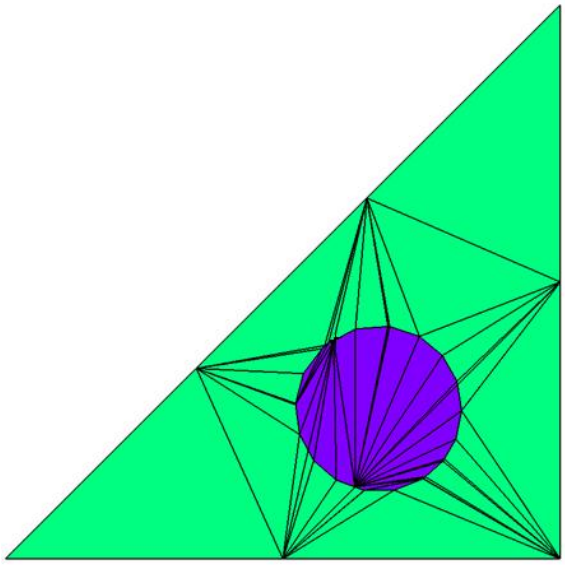


Figure 7.20 Coarse Triangular Mesh after Intersection, Loop Formed

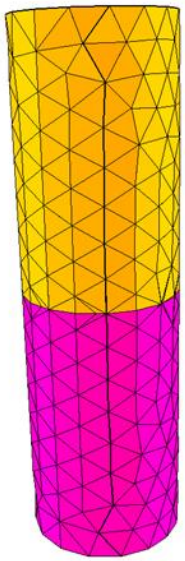


Figure 7.21 Cylinder after Intersection, Loop Formed

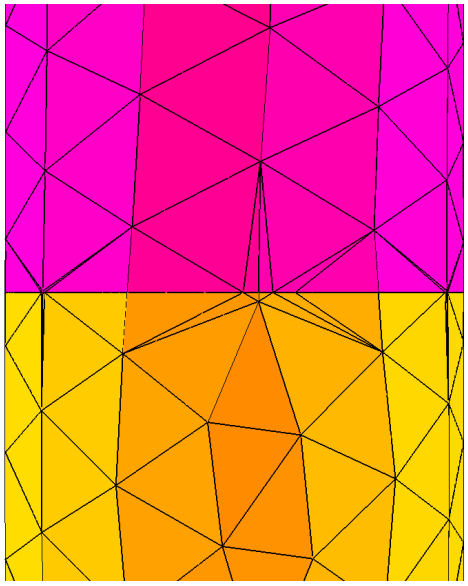


Figure 7.22 Close-up of Loop Formed

Circles, Big Loop Formed

The purpose of this example is to show the efficiency of the intersection tool and that it can be used for intersecting large meshes of differing resolutions. The mesh in Figure 7.23 is two intersecting spheres, blue and green. The green sphere has 56,324 nodes 112,644 triangles and the blue sphere has 108 nodes and 216 triangles. All of the fundamental types of intersections discussed in CHAPTER V were encountered during the intersection. The process of initializing the mesh, calculating the lines of intersection, inserting the lines into each mesh, and an application of the surface-painting algorithm took 1.2 second.

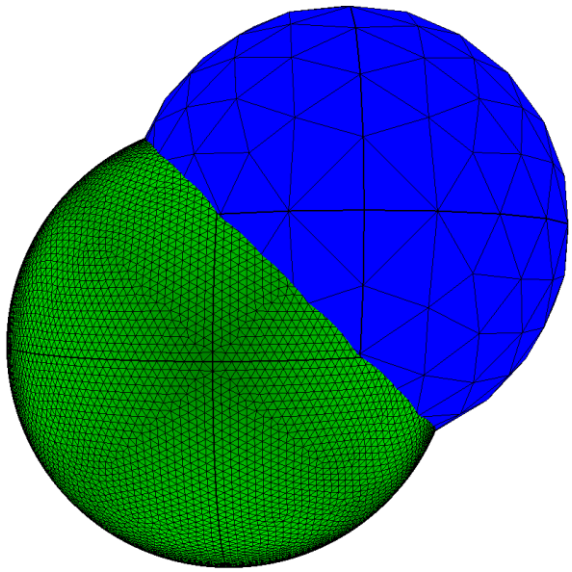


Figure 7.23 Intersecting Spheres

In Figure 7.24 and Figure 7.25 the results from the intersection and subsequent surface-painting can be seen. Both circles have had a closed loop of lines of intersection successfully inserted into the mesh.

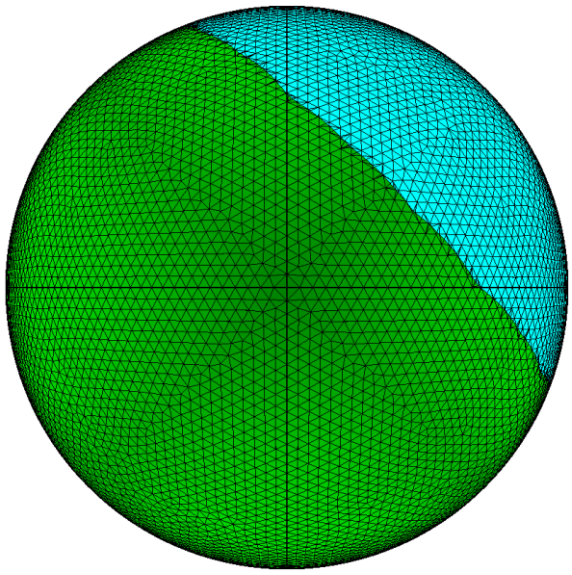


Figure 7.24 Results from Intersecting Spheres 1

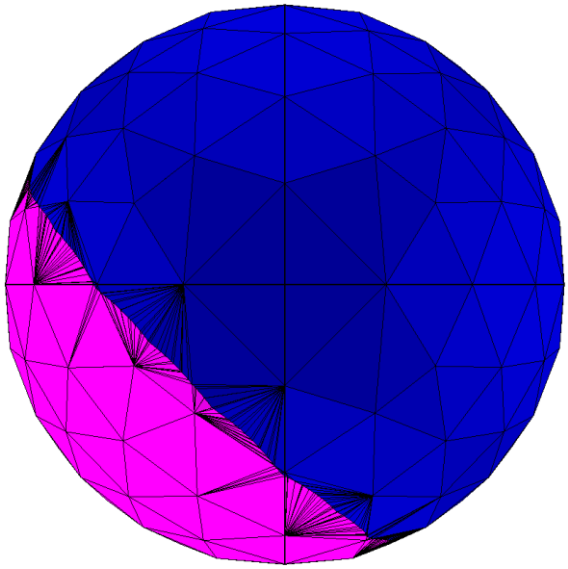


Figure 7.25 Results from Intersecting Spheres 2

If the desired result of the intersection was the whetted surface, the result would appear similar to the original geometry seen in Figure 7.23. In Figure 7.26 the result of

the intersection is shown where the desired geometry is the interior of the spheres. The results shown here, both above and below, exhibit no free-boundary edges.

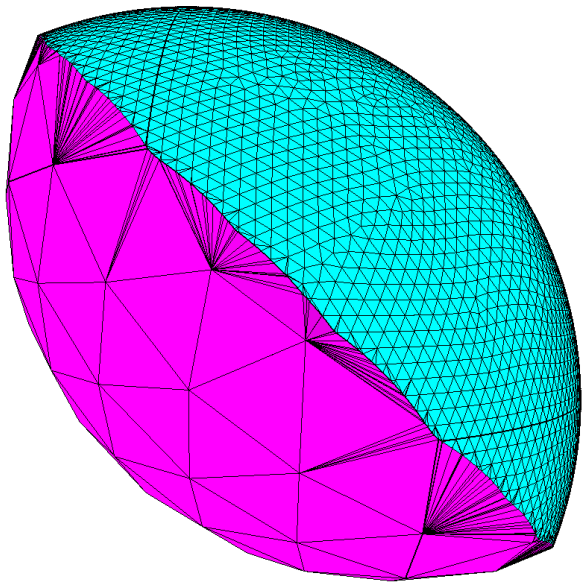


Figure 7.26 Interior of Spheres after Intersection

Flying Minnow

This example seeks to demonstrate all of the features of the intersection tool. In Figure 7.27 a sample model used to test mesh repair tools is shown. In this case, all of the geometry not relevant to this example has been removed to simplify the picture. What remains is the fuselage (purple) and the right wing (yellow). These two surfaces have different resolutions and while the fuselage's geometry is nearly isotropic around the intersection, the wing does not exhibit this quality. Near the leading edge, bottom right in Figure 7.28, some poor quality elements can be clearly seen. These elements were left as-is to demonstrate the robustness of the intersection tool. The fuselage has 2,138 nodes and 4,066 triangles and the wing has 936 nodes and 1,746 triangles.

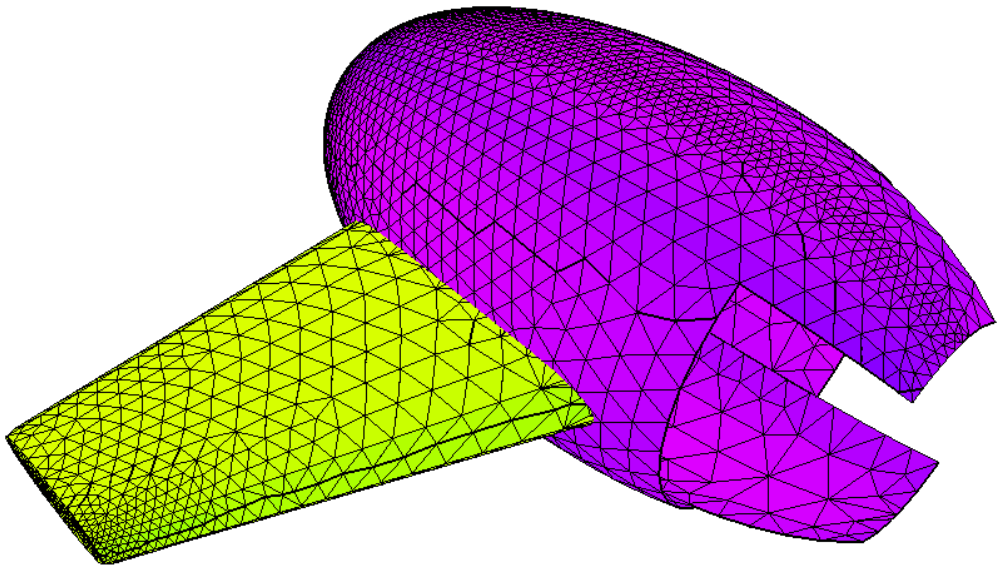


Figure 7.27 Right Wing Intersecting Fuselage of Flying Minnow

Figure 7.28 and Figure 7.29 shows the right wing alone in the frame and the fuselage alone in the frame, respectively, in order to demonstrate that these surfaces were not topologically adjacent before intersection.

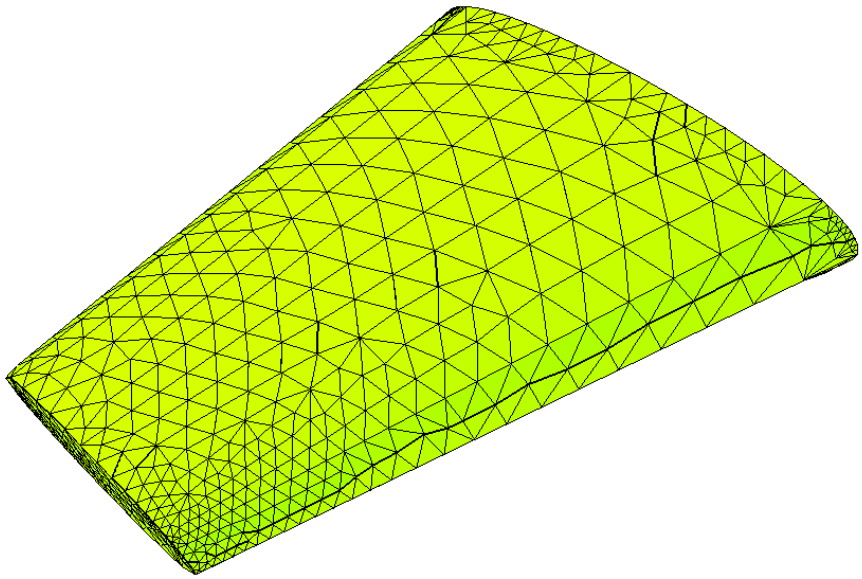


Figure 7.28 Right Wing of Flying Minnow

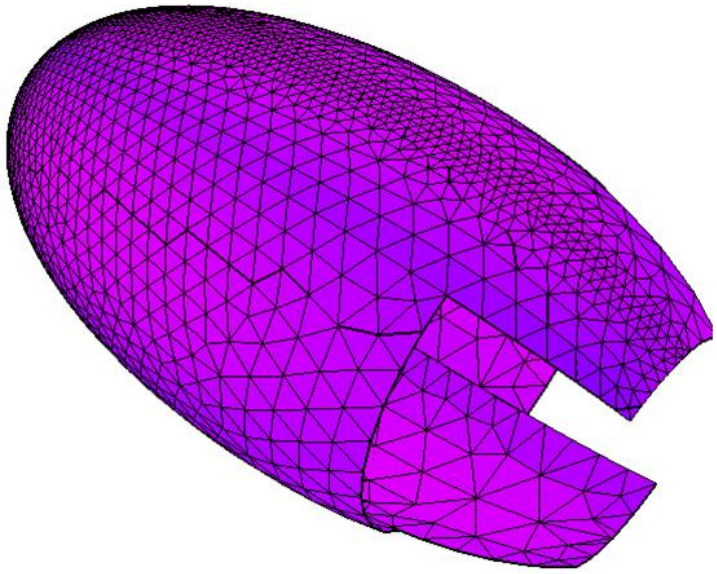


Figure 7.29 Fuselage of the Flying Minnow

Figure 7.30 shows the result of the intersection with both the wing and fuselage in the frame. The fuselage alone can be seen in Figure 7.31. The right wing alone can be seen in Figure 7.32. In each figure, it can be seen that the lines of intersection form a closes loop on the surface and therefore the surface-painting algorithm was able to separate the fuselage into three surfaces and the wing into two surfaces. The orange surface in these results is a discrete surface that is not topologically adjacent to the rest of the fuselage. Therefore the surface-painting algorithm, for which the EDGE_TEST() returns false for free-boundary edges, separated this surface from the rest of the fuselage. It is not an artifact of the intersection process.

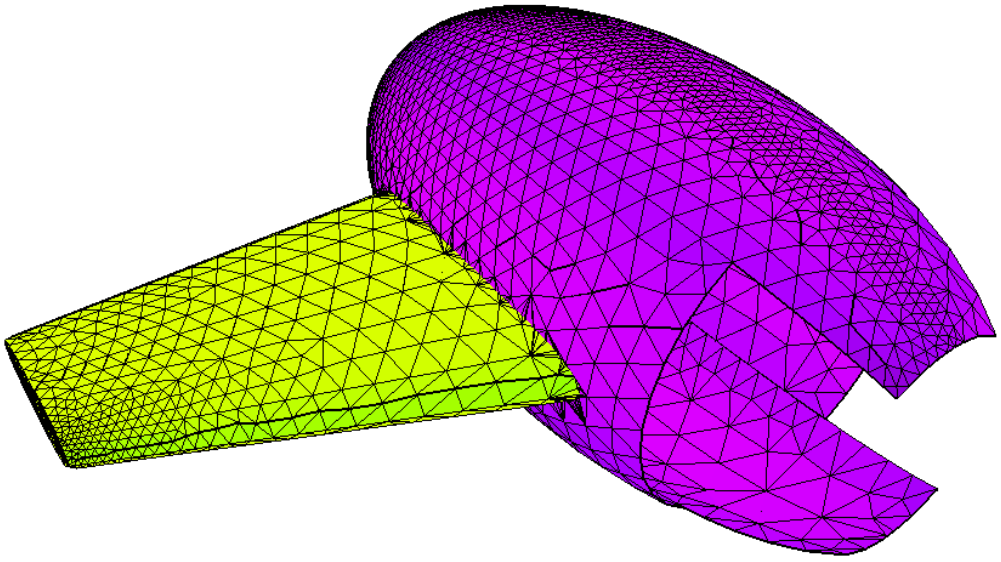


Figure 7.30 Results of Intersection, Flying Minnow

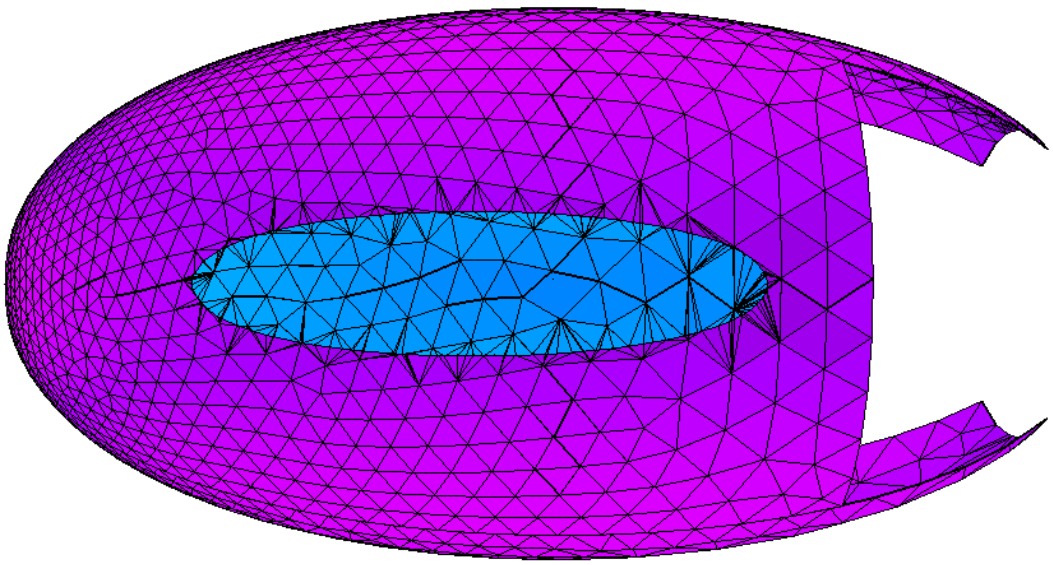


Figure 7.31 Result of Intersection, Fuselage of Flying Minnow

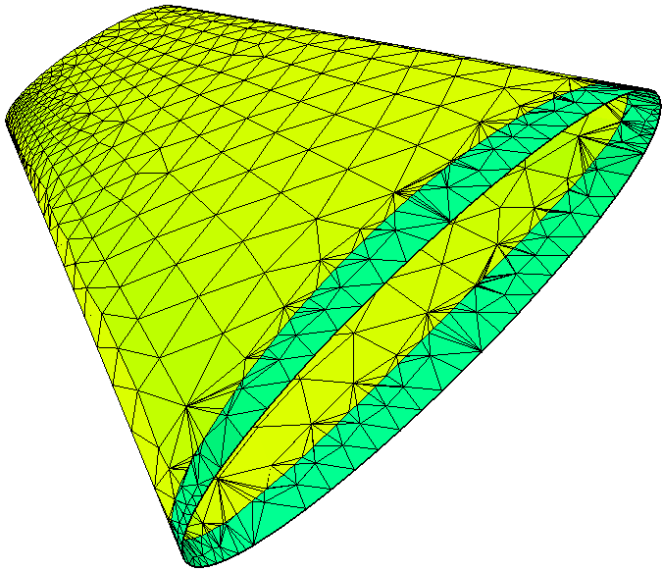


Figure 7.32 Result of Intersection, Right Wing of Flying Minnow

In Figure 7.33 an interior view of the model is given to show that the wing intersects the fuselage into the interior. Also, in both Figure 7.33 and Figure 7.34 the

free-boundary edges have been shaded bright yellow—as opposed to the yellow-green of the right wing. After the intersection process, the superfluous geometry on the wing and fuselage were deleted to produce the results seen in Figure 7.34. The free-boundary edges seen on the wing in Figure 7.33 are no longer present in Figure 7.34 and the wing is now topologically adjacent to the fuselage. 167 nodes and 668 triangles were added by the intersection of the two surfaces.

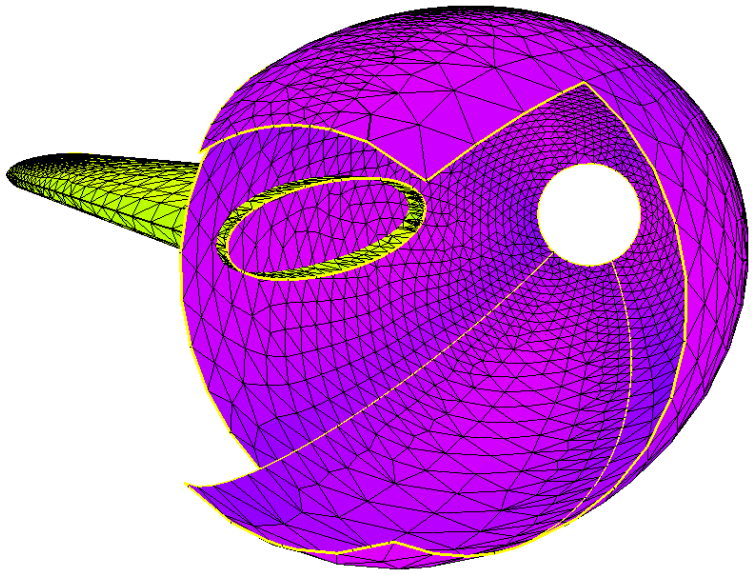


Figure 7.33 Interior view of Flying Minnow before Intersection

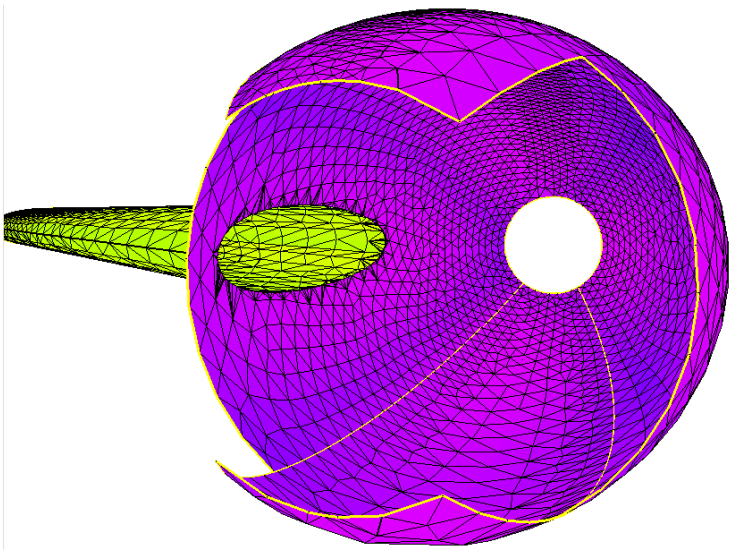


Figure 7.34 Interior view of Flying Minnow after Intersections

Isolated Boundary

In this section, the results from the repair of isolated boundaries are presented.

Edge Recovery

In this example, an open box (orange) with the opening toward an undulating surface (blue) is shown in Figure 7.35. First, results from the method of simply inserting an edge whole into a mesh are presented and discussed. Then results from the addition of edge splitting are shown.

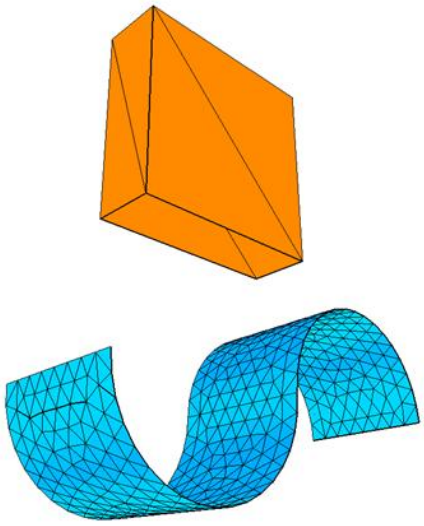


Figure 7.35 Open Box and Undulating Surface, Isolated Boundary

Whole Edge Recovery

In this example the results of whole edge extension and recovery are shown in Figure 7.36. The edges that were projected (orange surface) are large relative to the edge length present in the undulating surface (blue). When the large edge is recovered, the underlying shape of the undulating surface was distorted severely. In Figure 7.36 and

Figure 7.37 the undulating surface has large, distorted, self-intersecting, low-quality elements that do not follow the shape of the original mesh, which was nearly isotropic. In Figure 7.38 the open box and the surface created from the extended edges is shown. The quality of the elements of the extended-surface with respect to the open box is quite similar. Therefore, it can be concluded the whole-edge extension and recovery preserves the quality of the highest order, or longest edge present in the projected edges. However, with whole-edge extension and recovery there is the risk of creating large, distorted, self-intersecting, low-quality elements. The results shown in this example are unacceptable from a mesh repair point of view because the process intended to repair the model has instead created problems not associated with the tool.

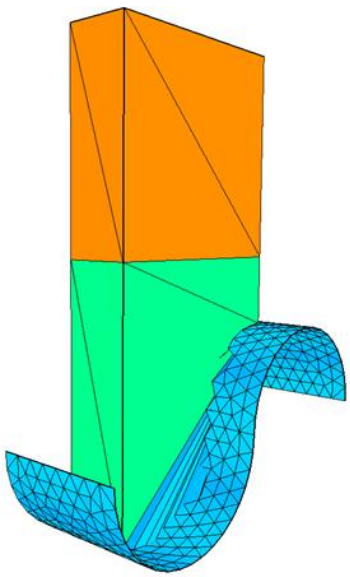


Figure 7.36 Results from whole-edge extension and recovery, Open Box and Undulating Surface

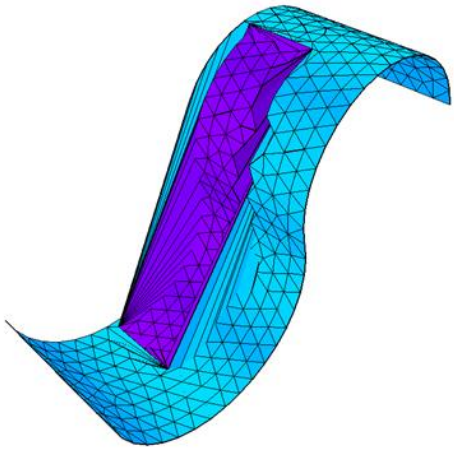


Figure 7.37 Results from whole-edge extension and recovery, Undulating surface

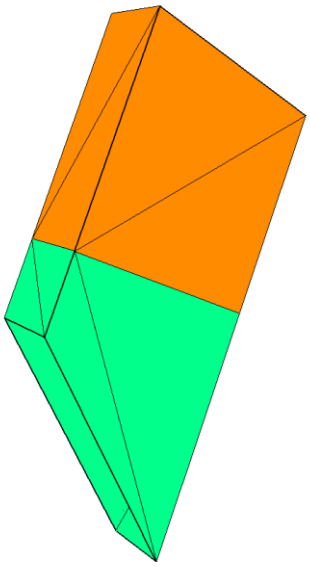


Figure 7.38 Results from whole-edge extension and recovery, Open Box and Surface produced from edge extension

Split Edge Recovery

In this example, the results from edge extension with splitting are presented. Figure 7.39 shows the results of projecting the edges of the open box (orange) onto the undulating surface below (blue). In Figure 7.40 and Figure 7.41 the individual components are show to demonstrate the advantage of the addition of edge splitting. The undulating surface no longer has large, distorted, self-intersecting, low-quality triangles that do not follow the shape of the original geometry. Instead, the only changes to the mesh are the splitting of edges. No edges were collapsed or reconnected. Since the relatively large edges of the open box were projected onto a surface with relatively small edges, the edges of the box were split many times. This effectively reduces the order, or length, of the largest edges present in the projection while maintain the lower order, or smaller, edges. The addition of edge splitting is seen as superior to whole-edge recovery due to the guarantee of topologically valid results. If the desired result were whole-edge recovery, the results from edge splitting could be altered by gluing nodes to achieve the results seen in Figure 7.36.

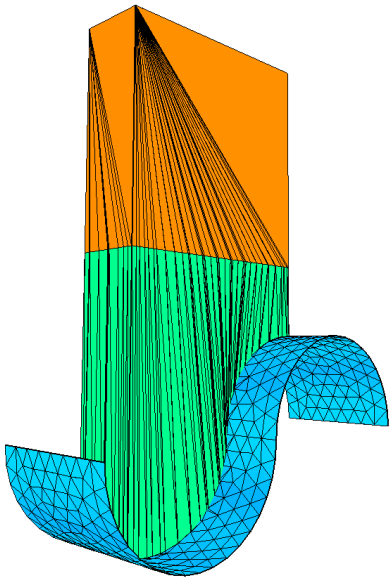


Figure 7.39 Results from edge splitting, Open Box and Undulating Surface

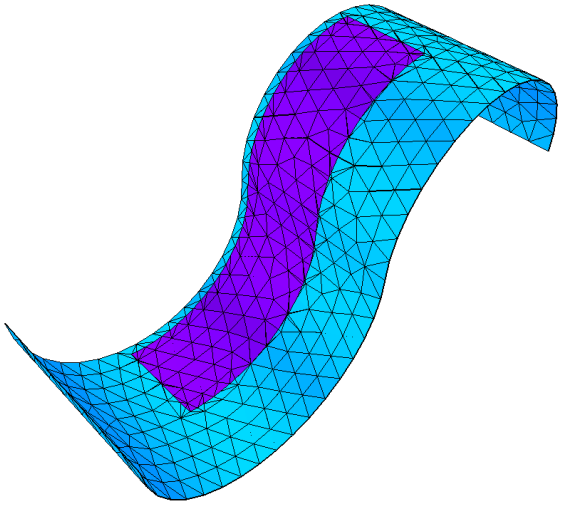
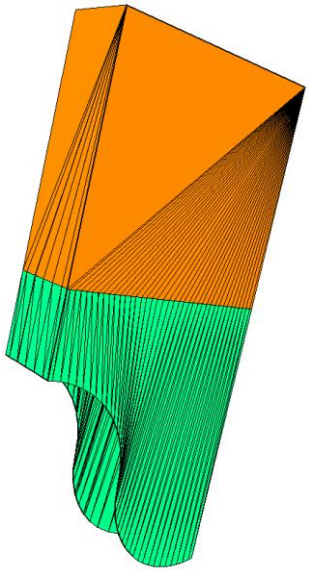


Figure 7.40 Results from whole-edge extension and recovery, Undulating Surface



s

Figure 7.41 Results from whole-edge extension and recovery, Open Box and surface produced from edge extension

Large Edge, Fine Surface/Closest Surface

The following example is shown to demonstrate two aspects of the isolated boundary repair tool: the tool only projects onto the nearest surface and the extension of a relatively large edge onto a finely meshed surface. The projection of a relative large edge onto a finely meshed surface demonstrates the robustness of the edge-tracking/edge-splitting aspect of the isolated boundary repair tool. In Figure 7.42, the wedge airfoil (green) has free-boundary edges that are to be projected onto the outer wall of the fuselage (pink, center). The inner wall of the fuselage (pink, left) shall remain unchanged, as it should, through the process of edge projection/splitting and recovery.

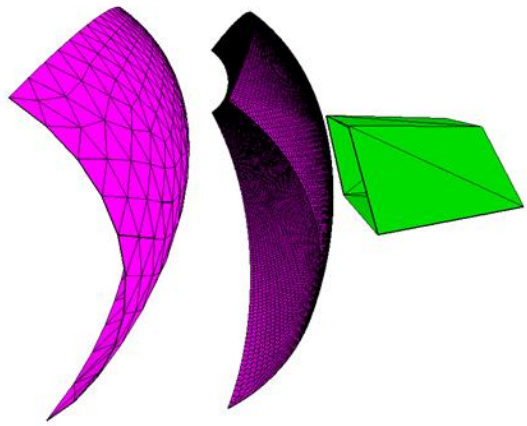


Figure 7.42 Wedge Airfoil and Outer/Inner Fuselage, Flying Minnow

The results from the projection, split, and subsequent filling of the gap can be seen in Figure 7.43. The wedge (green) airfoil's free-boundary edges have been projected onto the outer wall of the fuselage (yellow, center). The interior wall (pink, left) of the fuselage remains unchanged, as it should, since the outer wall is closer to the airfoil's

free-boundary edges. In Figure 7.44, the outer fuselage is shown alone in the frame. The edges that were projected onto the surface form a closed loop so the surface-painting algorithm was able to split the outer fuselage into to surfaces, the inner (blue) and outer (yellow). These surfaces are topologically adjacent, but distinct.

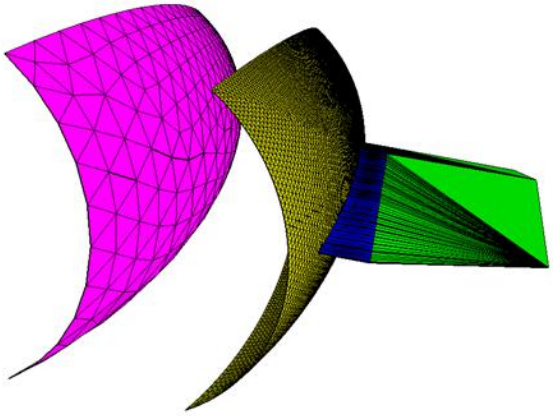


Figure 7.43 Results from edge projection, Wedge Airfoil and Outer/Inner Fuselage, Flying Minnow

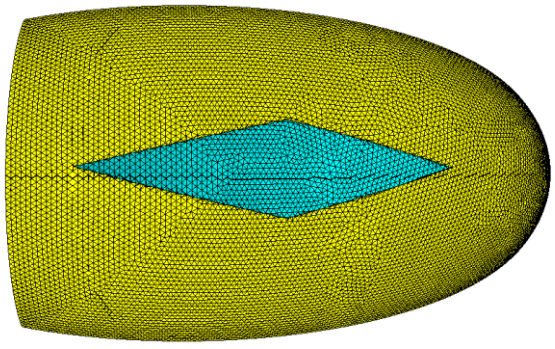


Figure 7.44 Results from edge projection, Outer Fuselage, Flying Minnow

The wedge airfoil (green) and the surface produced from the edge projection (blue) can be seen in Figure 7.45. The edges on the airfoil have been split many times to accommodate the smaller mesh resolution present on the surface of the fuselage. Also, it should be noted that the surface produced from the edge projection follows the curvature of the fuselage exactly. Since the mesh resolution is so fine, a close-up of the leading edge can be seen in Figure 7.46.

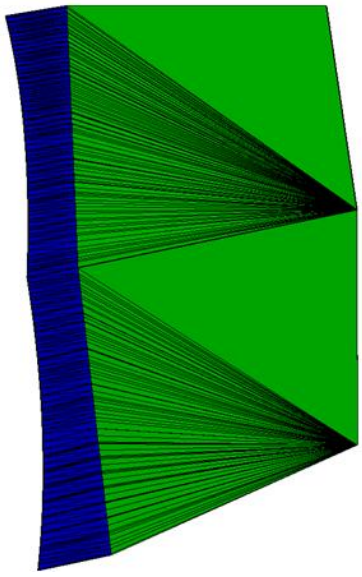


Figure 7.45 Results from edge projection, Wedge Airfoil, Flying Minnow

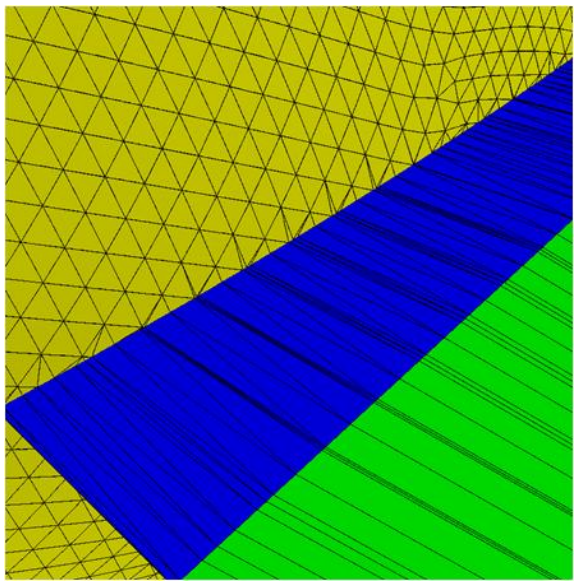


Figure 7.46 Results of edge projection, Wedge Airfoil and Outer Fuselage, Flying Minnow, leading edge close-up

Sports Utility Vehicle

This example seeks to demonstrate all of the aspects of the isolated boundary repair tool. In Figure 7.47 a model of a sports utility vehicle [29] is shown. This model was created for the purposes of graphical rendering and therefore is not watertight. One of the major imperfections present in this model is the gap near the driver's side mirror (Figure 7.48 and Figure 7.49).

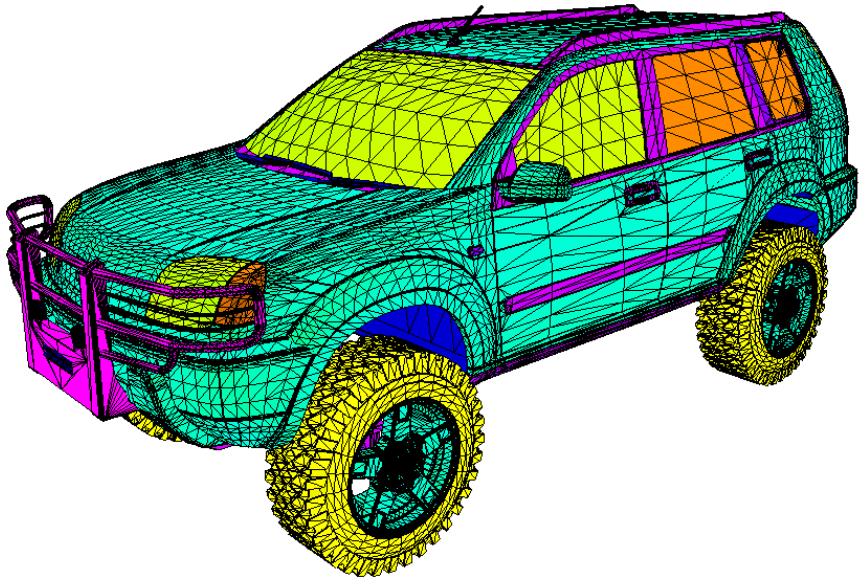


Figure 7.47 SUV Model

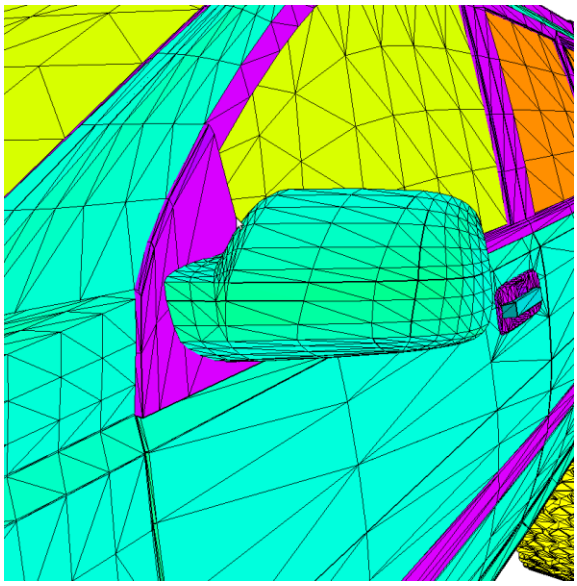


Figure 7.48 Close-up View of Problem Area in SUV Model

In Figure 7.48, a close-up of the driver's side rear-view mirror can be seen. In order to simplify presentation, all of the geometry not relevant to this tool has been removed. In Figure 7.49 and Figure 7.50, the gap between the rear-view mirror and the body (purple surface in Figure 7.48) of the SUV can be seen.

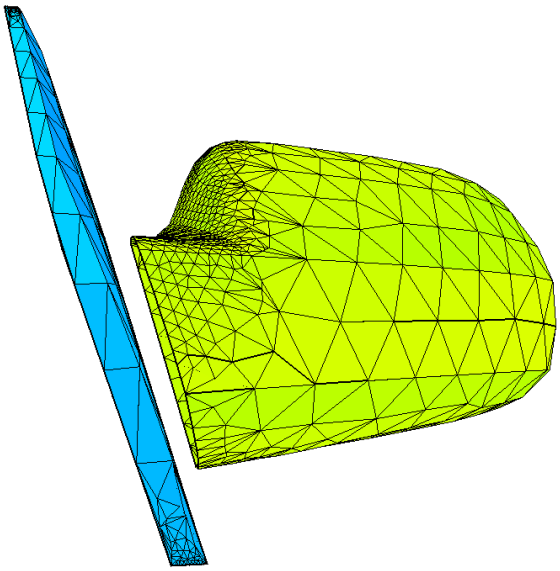


Figure 7.49 Isolated Rear view Mirror and Vehicle Trim of SUV Model

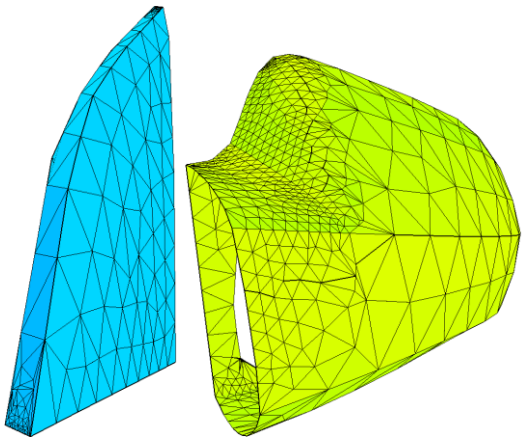


Figure 7.50 Isometric View of Vehicle Trim (left-blue) and Rear-view Mirror (right-green)

In Figure 7.51, the results of the isolated boundary repair tool can be seen. The free-boundary edges on the mirror (yellow) have been projected onto the nearby body (blue) and split. These edges were then used to create the surface that fills the gap (orange).

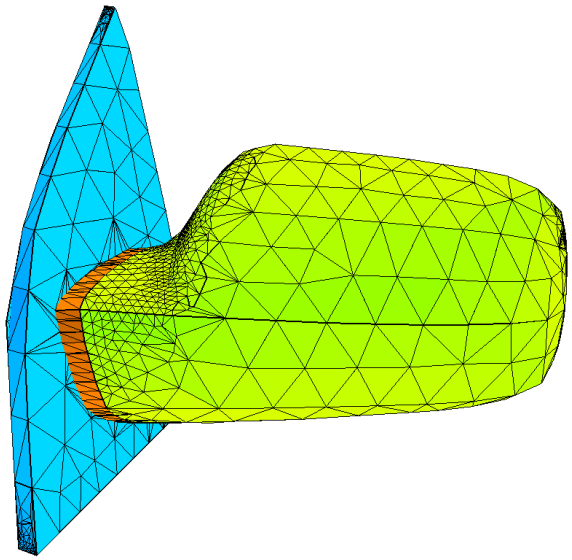


Figure 7.51 Result of Isolated Boundary Repair, Rear-view Mirror and Vehicle Body Near Mirror

In Figure 7.52, the nearby body is shown alone in the frame (blue) and the surface now defined by the projected edges (yellow-green) can be seen. In Figure 7.53, the mirror (yellow) and the surface created to fill the gap (orange) can be seen alone in the frame. The edges of the mirror that were projected onto the body create a closed loop. This allowed the surface-painting algorithm to separate the body panel into two surfaces, interior (yellow-green) and exterior (blue). These two surfaces are distinct but remain topologically adjacent.

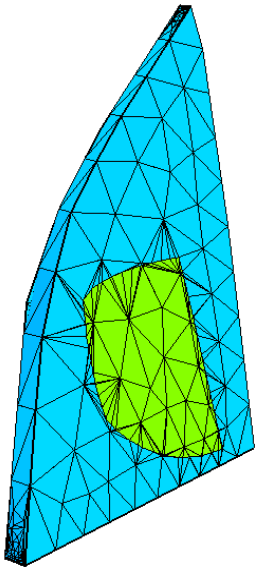


Figure 7.52 Results of Isolated Boundary Repair, Nearby Body, SUV Model

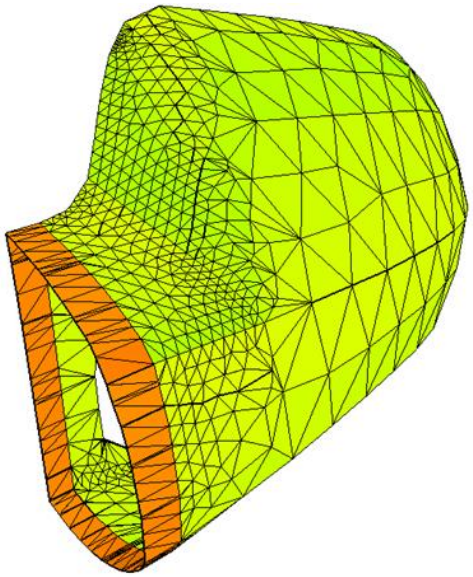


Figure 7.53 Result of Isolated Boundary Repair, Rear-view Mirror and Additional Surface, SUV Model

In Figure 7.51, Figure 7.52, and Figure 7.53 the results of the isolated boundary repair tool can be seen. These figures show the surface that is generated to repair the gap

present near the isolated boundaries as well as the affected, nearby geometry. The surface created by the edges of the mirror and their projections is now topologically adjacent to the vehicle trim and the mirror so that no free-boundary edges are created through the process of repairing the geometry. Many of these types of gaps or intersections, of widely varying scale, are present in this model. However, only the few that could be visually demonstrated have been presented here. For example, the driver's side window behind the mirror in Figure 7.48 has gaps and intersections with the trim around the window. These could be repaired, but the resulting geometry was unable to effectively shown here due to the size of the intersection and gaps, which are orders of magnitude smaller than the length scale on the window or trim.

Combined Application

In this section, examples that cannot be repaired using solely the intersection tool or solely the isolated-boundary repair tool will be presented. These examples will show that the tools can be used together to repair imperfect geometry.

SUV Steering Column and Dash

In Figure 7.54 the driver's side of the interior of the SUV [29] in Figure 7.47 can be seen. The geometry relevant to this example, the dash and steering column, can be seen alone and isolated in Figure 7.55 and Figure 7.57 respectively—where all of the irrelevant geometry has been removed to simplify presentation. In Figure 7.55 and Figure 7.56, the steering column can be seen intersecting the dash. However, the intersection is only partial and a gap is present between the bottom of the steering column and the dash (Figure 7.56, right).

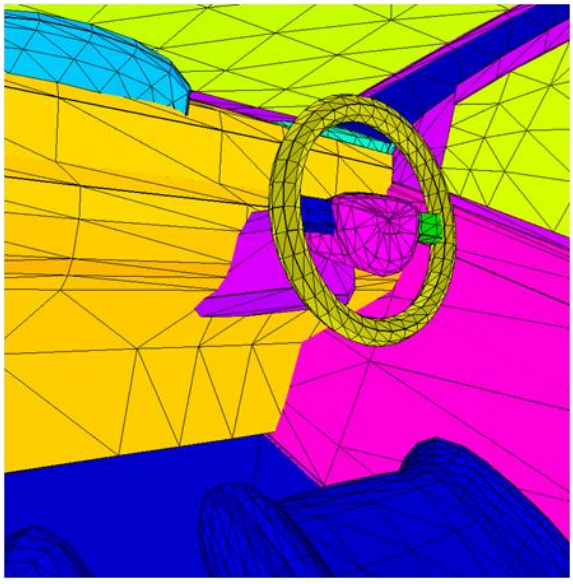


Figure 7.54 SUV Driver's Side

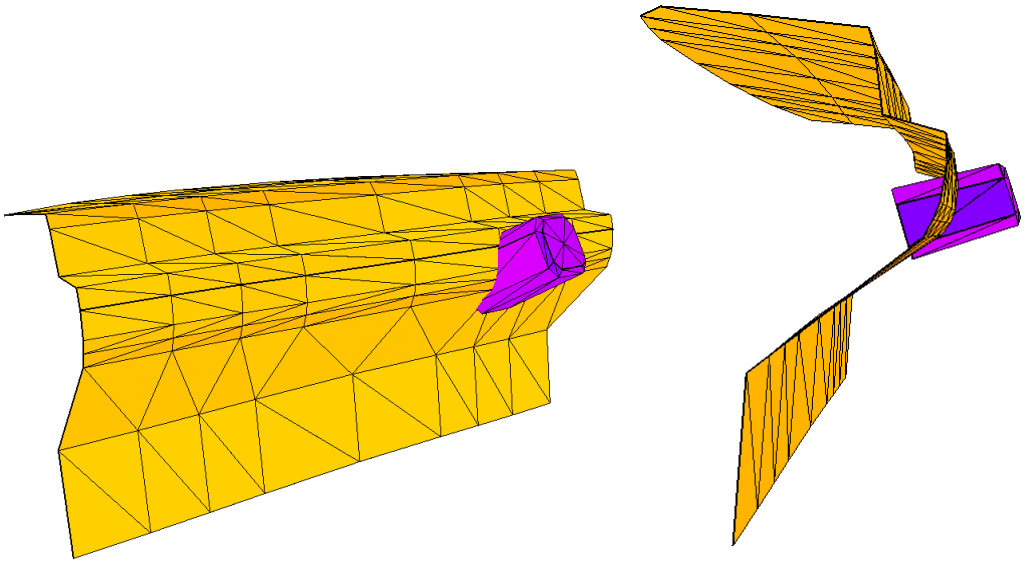


Figure 7.55 SUV Steering Column and Dash, Alone; Isometric (left), Side (right)

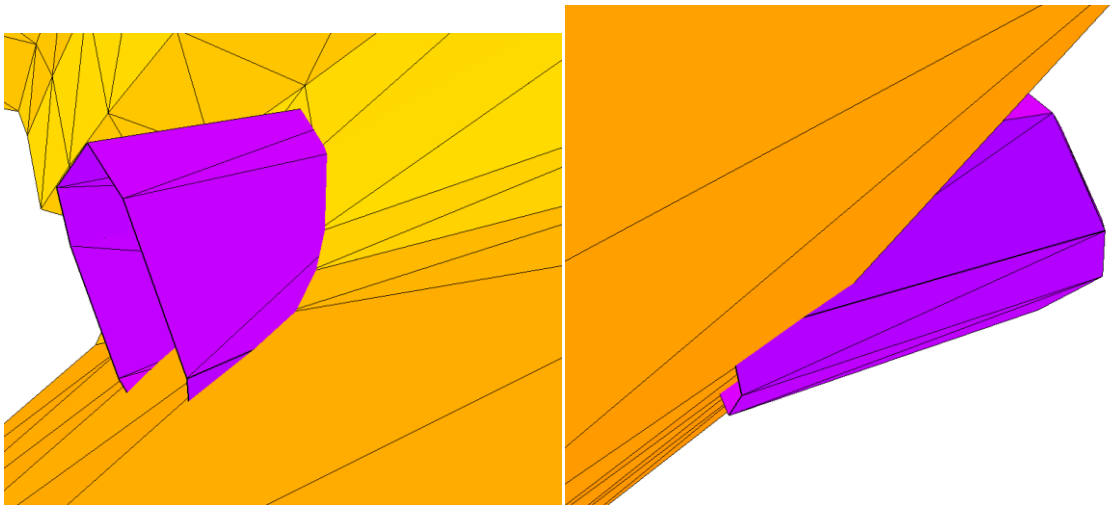


Figure 7.56 SUV Steering Column and Dash, Alone; Close-up of Intersection (left) and Gap (right)

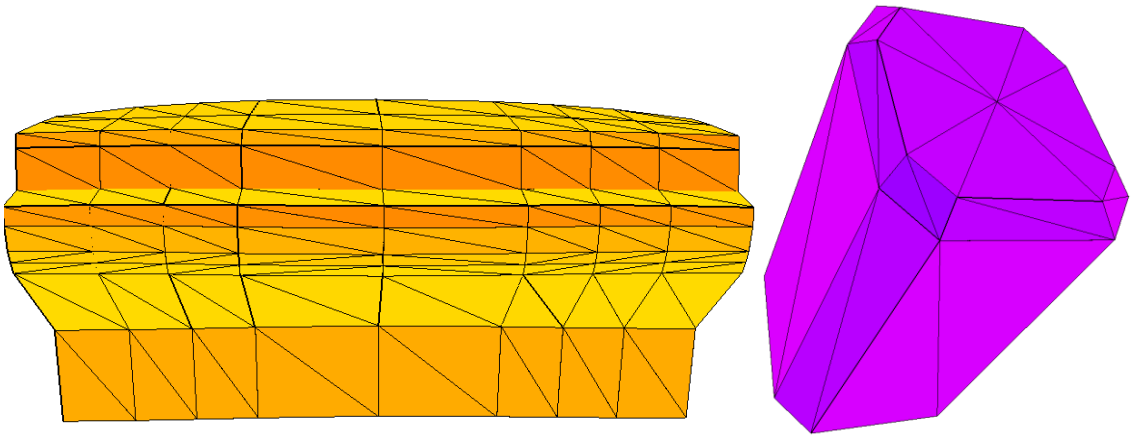


Figure 7.57 SUV Steering Column (right) and Dash (left), Isolated

To start the repair process, the dash and the steering column must first be intersected. Once that is complete, the undesired geometry will be removed and then the isolated boundaries will be repaired. Then the remainder of the undesired geometry will be removed to arrive at a fully repaired state. First, the results from the intersection: they can be seen in Figure 7.58 and Figure 7.59. Since the intersection was only partial, the dash surface was not broken into any further pieces by the surface painting algorithm (Figure 7.59, left). However, the steering column was broken into two surfaces since the intersection formed a closed loop with the free-boundary of the steering column (Figure 7.59, right). Once the undesirable geometry on the steering column (yellow surface, Figure 7.59, right) was removed and then the process of repairing the isolated boundaries (seen in Figure 7.56, right) could begin.

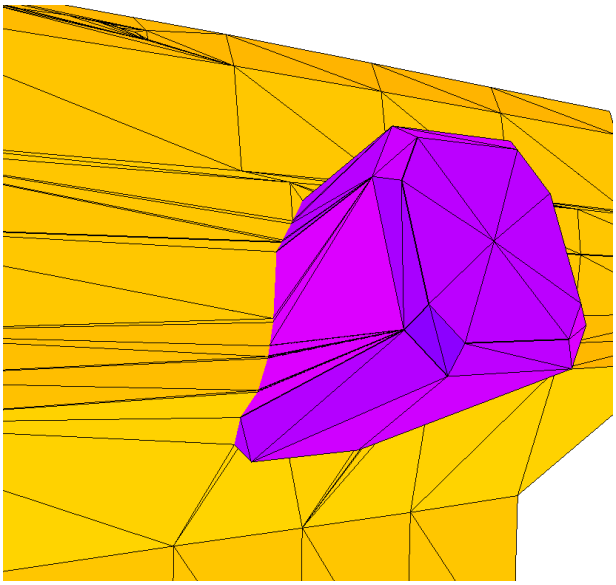


Figure 7.58 SUV Steering Column and Dash, Intersection Results

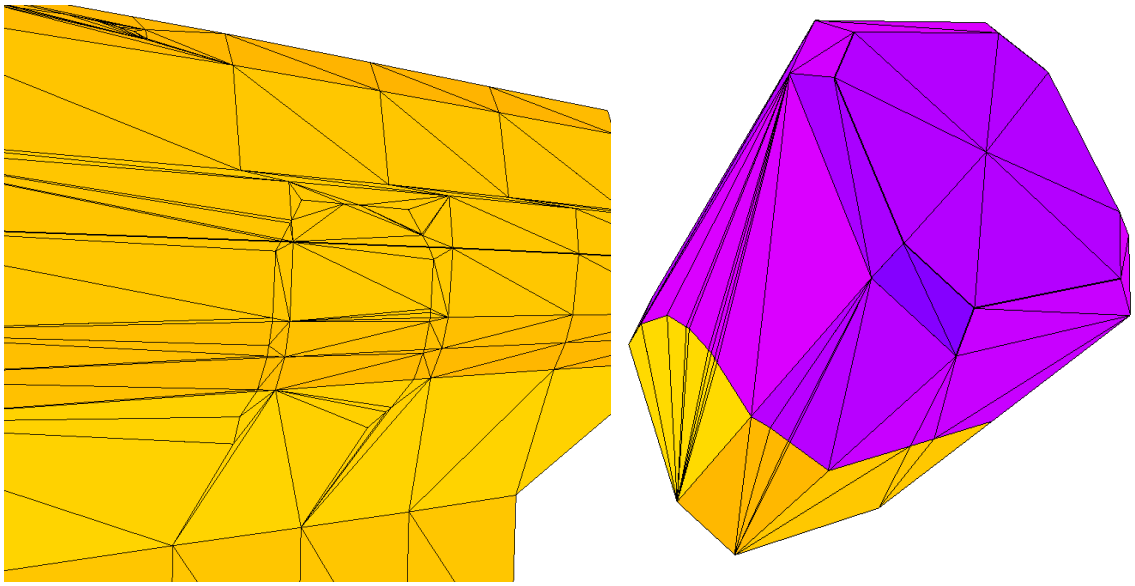


Figure 7.59 SUV Steering Column and Dash, Intersection Results, Isolated

In Figure 7.60 the results from the isolated-boundary repair can be seen. The small gap at the bottom of the steering column has been filled with elements (blue

surface, Figure 7.60) that are topologically adjacent to both the dash and steering column. In Figure 7.61 the dash and steering column can be seen isolated from each other. This figure shows the closed loop formed on the dash by the combination of the intersection and edge projection. Since a closed loop is formed, the dash was broken into two surfaces using the surface painting algorithm. The results from this removal can be best seen in Figure 7.60, right, which shows the repair location from the rear.

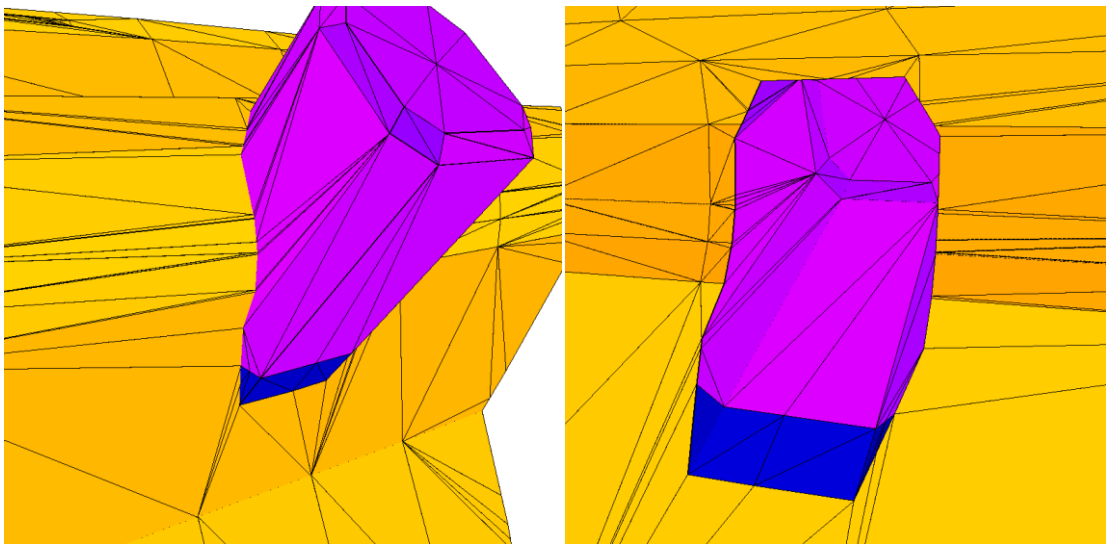


Figure 7.60 SUV Steering Column and Dash, Alone, Isolated Boundary Repaired, Front Isometric (left), Rear Isometric (right)

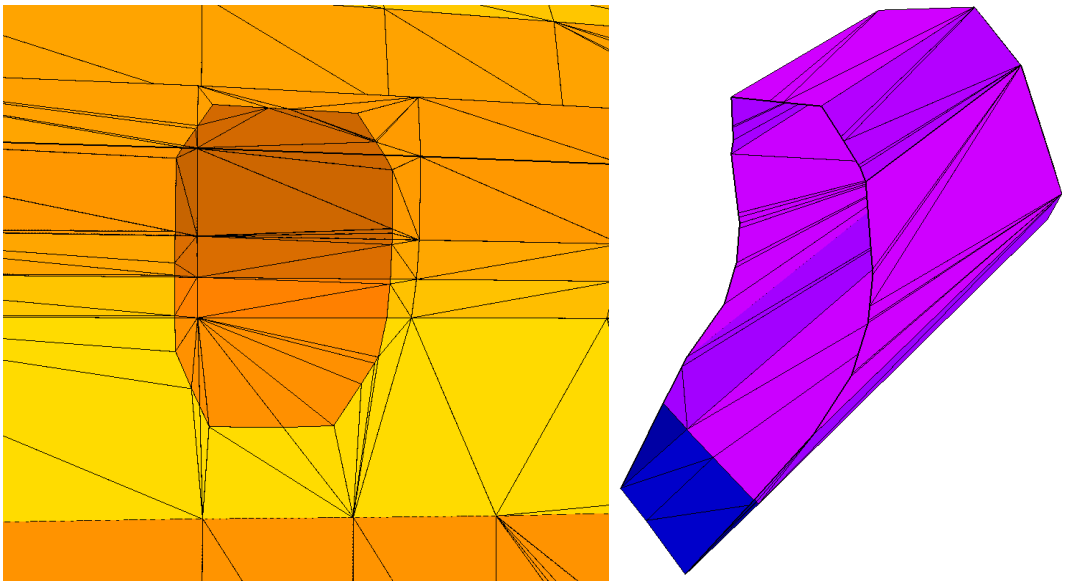


Figure 7.61 SUV Steering Column and Dash, Isolated; Isolated-Boundary Repaired; Dash (left), Steering Column (right)

This example shows the combined results of the repair of a partial intersection and isolated boundaries. The result is topologically valid and exhibits no free-boundary edges at the intersection location or at the edge-projection location. Once the undesired geometry is removed, the result is also manifold and the steering column is now topologically adjacent to the dash.

CHAPTER VIII

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 the tools repair the models while maintaining small features and curvature present in the original data. The tools were shown to be robus 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.

Summary of Contributions

The primary contributions of this work are:

Repair of Discrete Mesh Intersections

A tool to repair intersecting triangular meshes was 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, i.e. the intersection was removed by replacing it with non-manifold edges.

A surface-painting algorithm that is bounded by surface boundary edges, free-boundary edges, and non-manifold edges was used to "break out" the surfaces that are bounded in whole or in part by the newly created non-manifold edges. The non-desired geometry can then be removed to arrive at fully repaired geometry that is manifold and free of boundary edges at the intersection. This tool repairs intersecting discrete geometry in-place, i.e. without having to re-mesh the intersecting surfaces. Results show that this tool effectively, efficiently, and accurately captures and repairs the intersection of discrete surfaces.

Repair of Isolated Boundaries

Methods for projecting isolated free-boundaries onto nearby discrete surfaces were developed. An octree was used for the purposes of casting rays from the isolated free-boundaries. The rays represent a projection direction and are used to find the nearest surface in order to fill the gap between the isolated free-boundary edges and the nearest surface. The use of the octree once again reduced the number of candidates required to consider for intersection with the ray. Robust methods for determining the intersected entity (node, edge or triangle) were also implemented. Once an intersection was found,

the proper entity was split, if needed. An edge-tracking algorithm that implements controlled point-line projections is used to find the edges that are to be split in order to recover the original edge as a set of connected line-segments. After the edges were projected onto the surface, they were split for the purposes of maintaining the curvature in the projected-upon surface as well as small features. This addition of edge splitting to the process of edge recovery is seen as an improvement over whole-edge recovery. Once the edges in the projected-upon surface were split, the edge being projected is split so that it has the same number of nodes required to represent it in the projected-upon surface due to the required edge splitting. With this set of nodes a straightforward method was implemented to fill the gap between the free-boundary edges and the nearby surface.

A surface-painting algorithm that is bounded by surface boundary edges, free-boundary edges, and non-manifold edges was used to "break out" the surfaces that are bounded in whole or in part by the newly created non-manifold edges. The non-desired geometry can then be removed to arrive at fully repaired geometry that is manifold and free of boundary edges at the projected edges. Additionally in areas where the distance between the free-boundary edges and the projected-upon surface is small, edges can be collapsed. This allows the final product to resemble the free-boundary edges being glued to the surface instead of projected. Results show that this tool effectively, efficiently, and accurately repairs isolated boundaries by filling the gap after projecting edges onto a nearby surface.

Future Directions/Work

This work is a step towards automated mesh repair and addresses two specific problems not previously addressed. Further research should include methods in which

these fundamental operations are applied automatically to repair geometry. This could include applying these tools in cases where the desired result of the tool is known, but the number of applications is impractical for semi-autonomous repair. Applying these tools in this fashion could be done presently with the existing API. However, the most useful development would be a set of algorithms that could automatically remove unwanted geometry after the intersections or isolated boundaries are repaired. Since the step of manually removing the unwanted geometry is the only step in the process that requires user intervention, some level of automation in this area would significantly reduce the time required to repair geometries with a large amount of imperfections.

Repairing intersections and isolated boundaries can now be complete semi-autonomously; however further research could improve the results of the intersections and projections. For example, once the intersection/projection has been performed the mesh might contain a large amount of high-aspect-ratio triangles and very small edges. Since the surfaces that were once distinct are topologically adjacent, mesh smoothing could be used along the lines of intersection/projection to improve the quality of the mesh near the intersection/projection. This would be useful if the user desired to use the repaired mesh as a background mesh for mesh generation purposes. Increasing the element quality would improve any mappings created and therefore higher quality meshes could be generated from the repaired and then improved mesh.

Even though the use of the octree has severely decreased the number of required number of candidates to test for intersection, a space and time complexity analysis could be done. This would allow comparisons for other types of spatial subdivision data structures like k-d trees and ADT's (alternating digital tree). However, for development

and simplicity's sake, having the elements and the nodes in one data structure proved useful and efficient.

REFERENCES

[1] Bischoff, Stephan, and Leif Kobbelt, "Structure Preserving CAD Model Repair," *Eurographics*, 24, no.3 (2005).

[2] Patel, Paresh S., Marcum, David L., and Remotigue, Michael R., "Stitching and Filling: Creating Conformal Faceted Geometry," *Proceedings of the 14 International Meshing Roundtable*, CA September, 2005.

[3] Chong, C. S., Kumar, A. Senthil, and H. P. Lee, "Automatic mesh-healing technique for model repair and finite element model generation," *Finite Elements in Analysis and Desing*, 43, (2007), pp. 1109-1119.

[4] Patel, Paresh S., Marcum, David L., and Remotigue, Michael R., "Automatic CAD model topology generation," *International Journal for Numerical Methods in Fluids*, 52 (2006), pp. 823-841.

[5] Weihe, Karsten and Thomas Willhalm, "Why CAD Data Repair Requires Discrete Algorithmic Techniques," Fakultät für Mathematik und Informatik, Universität Konstanz, May 1998.

[6] Park, Sang C., "Triangular mesh intersection," Department of Information & Systems Engineering, Ajou University, August 2004.

[7] Aftosmis, M. J., Berger, M. J., and J. E. Melton, "Robust and Efficient Cartesian Mesh Generation for Component-Based Geometry," U.S. Air Force Wright Laboratory / NASA Ames, CA.

[8] Lo, S. H., and W. X. Wang, "Finite element mesh generation over intersecting curved surfaces by tracing of neighbors," *Finite Element in Analysis and Design*, 41 (2005) pp. 351-370.

[9] Steinbrenner, J.P., Wyman, N.J., and J.R. Chawner, "Procedural CAD Model Edge Tolerance Negotiation for Surface Meshing," *Engineering with Computers*, 17, 2001, pp. 315-325.

[10] Tysell, Lars, "CAD Geometry Import for Grid Generation," SE-172 90, Swedish Defence Research Agency, Stockholm, Sweden.

[11] Wagner, Marc, Labsik, Ulf, and Günther Greiner, "Repairing Non-Manifold Triangle Meshes Using Simulated Annealing," *International Journal of Shape Modeling*, 9, no. 2 (2003), pp. 137-153.

[12] Pernot, Jean-Philippe, Moraru, George, and Philippe Véron, "Filling holes in meshes using a mechanical model to simulate the curvature variation minimization," *Computers and Graphics*, 30 (2006), pp. 892-902.

[13] Liepa, Peter, "Filling Holes in Meshes," *Eurographics Symposium on Geometry Processing*, 2003.

[14] Barequet, Gill and Micha Sharir, "Filling Gaps in the Boundary of a Polyhedron," *Computer-Aided Geometric Design*, 12, no. 2 (1995), pp. 207-229.

[15] Levin, Adi, "Filling an N-sided hole using combined subdivision schemes," Tel Aviv University, June 1999.

[16] Guo, Tong-Qiang, Li, Ju-Jun, Weng, Jian-Guang, and Yue-Ting Zhuang, "Filling Holes in Complex Surfaces Using Oriented Voxel Diffusion," *Proceedings: Fifth International conference on Machine Learning and Cybernetics*, Dalian, August 2006.

[17] Lorensen, W., and H. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *SIGGRAPH*, 1987.

[18] Samet, Hanan. 1990. "The Design and Analysis of Spatial data Structures." Reading: Addison-Wesley Publishing.

[19] Sagawa, Ryusuke, and Katsushi Ikeuchi, "Hole Filling of a 3D Model by Flipping Signs on Signed Distance Field in Adaptive Resolution," *IEEE Transactions on Patter Analysis and Machine Intelligence*, 30, no. 4 (2008).

[20] Hussain, Muhammad, Okada, Yoshihiro, and Koichi Niijima, "Efficient and Feature-Preserving Triangular Mesh Decimation," *Journal of WSCG*, 12, no. 1-3 (2004).

[21] Bischoff, Stephan, Pavic, Darko, and Leif Kobbelt, "Automatic Restoration of Polygonal Models," Computer Graphics Group, RWTH Aachen University.

[22] Möller, Tomas, "A Fast Triangle-Triangle Intersection Test," *Journal of Graphics Tools*, 2, no. 2 (1997).

[23] Mahovsky, J. and Brian Wyvill, "Fast Ray-Axis Aligned Bounding Box Overlap Tests with Plücker Coordinates

[24] Gellert, W., Gottwald, S., Hellwich, M., Kästner, H, and H. Knstner (Eds), *VNR Concise Encyclopedia of Mathematics, 2nd ed*. New Yor, Van Nostrand Reinhold, pp. 541-543, 1989.

[25] George, P. L., Hecht, F., and E. Saltel, "Automatic mesh generator with specified boundaries," *Computer Methods in Applied Mechanics and Engineering*, 92 (1991), pp. 269-288.

[26] Lawson, Charles L., "Properties of n-dimensional triangulations", *CAGD* 3, no. 4 (1986), pp. 231-246.

[27] Edelsbrunner, H. and Ernst Peter Mücke, "Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms", University of Illinois at Urbana-Champain, 1990.

[28] Karamete, B. Kaan, Garimella, Rao V., and Mark S. Shepard, "Recovery of an arbitrary edge on an existing surface mesh using local mesh modifications," *International Journal for Numerical Methods in Engineering*, 50 (2001), pp. 1389-1409.

[29] Weier, J., "Nissan XTrail 4x4 [Tuned]," http://www.3dcadbrowser.com/preview.aspx?ModelCode=12486, August 2007.

[30] Remotigue, Mike, Blades, Eric and David McLaurin, "Topologically Corrected Discrete and Sparse Data," Final Project Report.

[31] Thilmany, Jean, "Translation time: design and analysis programs don't speak the same language; getting around that can be costly.," *Mechanical Engineering-CIME*, 2006.

[32] Lee, Y. K., Lim, Chin K., Ghazialam, H., Vardhan, H., Edlund, E., "Surface Mesh Generation for Dirty Geometries by Shrink Wrapping using Cartesian Grid Approach," *Proceeding of the 15th International Meshing Round Table*, Birmingham, Alabama, September 2006.

[33] Cebral, J. R., Camelli, F. E., Löhner, R., "A feature-preserving volumetric technique to merge surface triangulations," *International Jounal for Numerical Methods in Engineering*, 55 (2002), pp. 177-190.

[34] Gaither, J. A., Marcum, D. L., and B. Mitchell, "SolidMesh: A Solid Modeling Approach to Unstructured Grid Generation," *Numerical Grid Generation in Computational Field Simulations, Proceedings of the 7th International Grid Generation Conference*, Whistler, British Columbia, September 2000.

[35] Marcum, D. L., "Parameterization of Tesselated Surfaces for Surface Mesh Generation," *SIMA Conference on Geometric Design*, Sacramento, CA, November 2001.

[36] Blades, E. L., Remotigue, M. G., and Marcum, D. L., "Utilizing Topological Discrete Triangulated Data within SolidMesh," *10th ISGG Conference on Numerical Grid Generation*, Crete, Greece, 2007.

[37] Goos, J, and J. Hartmanis. 1988. Efficient Structures for Geometric Data Management. Springer-Verlag.

[38] Sahni, Sartaj. 2005. Data Structures, Algorithms, and Applications in C++. Silicon Press.

[39] Petersson, N. Anders, and K. K. Chand, "Detecting Translation Errors in CAD Surfaces and Preparing Geometries for Mesh Generation," *Proceedings of the 10th International Meshing Roundtable*, CA, October 2001.

[40] Möller, T. and Ben Trumbore, "Fast, Minimum Storage Ray/Triangle Intersection", Chalmers University of Technology, 1997.

APPENDIX A

SUPPLEMENTARY C++ SOURCE CODE

MATH UTILITIES

These utilities use the *#define* pre-processing directive available in C++. These utilities are used to make code more readable.

```
//Useful #defines
#define GRX_CROSS(dest,v1,v2)    \
  dest[0]=v1[1]*v2[2]-v1[2]*v2[1]; \
  dest[1]=v1[2]*v2[0]-v1[0]*v2[2]; \
  dest[2]=v1[0]*v2[1]-v1[1]*v2[0];

#define GRX_DOT(v1,v2) (v1[0]*v2[0]+v1[1]*v2[1]+v1[2]*v2[2])

#define GRX_ADD(dest,v1,v2)              \
  dest[0]=v1[0]+v2[0];                   \
  dest[1]=v1[1]+v2[1];                   \
  dest[2]=v1[2]+v2[2];

#define GRX_SUB(dest,v1,v2)              \
  dest[0]=v1[0]-v2[0];                   \
  dest[1]=v1[1]-v2[1];                   \
  dest[2]=v1[2]-v2[2];

#define GRX_MUL(dest,v,t)                \
  dest[0] = v[0] * t;                    \
  dest[1] = v[1] * t;                    \
  dest[2] = v[2] * t;

#define GRX_DIV(dest,v,t)                \
  dest[0] = v[0] / t;                    \
  dest[1] = v[1] / t;                    \
  dest[2] = v[2] / t;

#define GRX_NORM2(v1)                    \
  (v1[0]*v1[0] + v1[1]*v1[1] + v1[2]*v1[2])

#define GRX_NORM(v1)                     \
  sqrt(GRX_NORM2(v1))

#define GRX_NORMALIZE(dest,v1)                              \
  dest[2] = sqrt(v1[0]*v1[0] + v1[1]*v1[1] + v1[2]*v1[2]); \
  dest[0] = v1[0] / dest[2];                         \
  dest[1] = v1[1] / dest[2];                         \
  dest[2] = v1[2] / dest[2];

#define GRX_SET(dest,v1)                 \
  dest[0] = v1[0];                       \
  dest[1] = v1[1];                       \
  dest[2] = v1[2];

#define GRX_TRAVEL(dest,orig,dir,t)      \
```

```
  dest[0] = orig[0] + t*dir[0];                \
  dest[1] = orig[1] + t*dir[1];                \
  dest[2] = orig[2] + t*dir[2];

#define GRX_ZERO(dest)                      \
  if(dest[0] < epsilon && dest[0] > -epsilon)   \
    dest[0] = 0.;                    \
  if(dest[1] < epsilon && dest[1] > -epsilon)   \
    dest[1] = 0.;                    \
  if(dest[2] < epsilon && dest[2] > -epsilon)   \
    dest[2] = 0.;

#define GRX_PI 3.14159265358979323846264338327950288419716939937510

#define GRX_TRANSFORM(dest,t,x)                          \
  dest[0] = t[0][0]*x[0] + t[0][1]*x[1] + t[0][2]*x[2];     \
  dest[1] = t[1][0]*x[0] + t[1][1]*x[1] + t[1][2]*x[2];     \
  dest[2] = t[2][0]*x[0] + t[2][1]*x[1] + t[2][2]*x[2];
#endif
```

RAY TRIANGLE INTERSECTION C++ SOURCE CODE

Adapted from [40]

```
int
intersect_triangle
(double orig[3], double dir[3],
 double vert0[3], double vert1[3], double vert2[3],
 double *t, double *u, double *v)
{
  double edge1[3], edge2[3], tvec[3], pvec[3], qvec[3];
  double det,inv_det;
  double epsilon = numeric_limits<double>::epsilon();

  //find vectors for two edges sharing vert0
  GRX_SUB(edge1,vert1,vert0);
  GRX_SUB(edge2,vert2,vert0);

  //begin calculating determinant - also used to calculate U parameter
  GRX_CROSS(pvec,dir,edge2);

  //if determinant is near zero, ray lies in plane of triangle
  det = GRX_DOT(edge1,pvec);

  if(det > -epsilon && det < epsilon)
    return 0;

  inv_det = 1.0/det;

  //calculate distance from vert0 to ray origin
  GRX_SUB(tvec,orig,vert0);

  //calculate U parameter and test bounds
  *u = GRX_DOT(tvec,pvec) * inv_det;
  if(*u < 0.0 || *u > 1.0)
    return 0;

  //prepare to test V parameter
  GRX_CROSS(qvec,tvec,edge1);

  //calculate V parameter and test bounds
  *v = GRX_DOT(dir,qvec) * inv_det;
  if(*v < 0.0 || *u + *v > 1.0)
    return 0;

  //calculate t, ray intersects triangle
  *t = GRX_DOT(edge2,qvec) * inv_det;

  return 1;
}
```

ROTATION TRANSFORMATION MATRIX CALCULATION C++ SOURCE CODE

```
void
grx_construct_transformation_matrix
(double *axis,
 double angle,
 double *origin,
 double transformation[][3],
 double epsilon)
{
  //This function will construct a transformation matrix that will
  // rotate something about the "axis" a total of "angle" at "origin"
  double norm_axis[3];
  GRX_NORMALIZE(norm_axis,axis);

  double c, s, ux, uy, uz, uxx, uxy, uxz, uyy, uyz, uzz;

  double dc1 = 1.;

  c = cos(angle);
  s = sin(angle);
  ux = norm_axis[0];
  uy = norm_axis[1];
  uz = norm_axis[2];
  uxx = norm_axis[0]*norm_axis[0];
  uxy = norm_axis[0]*norm_axis[1];
  uxz = norm_axis[0]*norm_axis[2];
  uyy = norm_axis[1]*norm_axis[1];
  uyz = norm_axis[1]*norm_axis[2];
  uzz = norm_axis[2]*norm_axis[2];

  transformation[0][0] = uxx + (dc1 - uxx)*c;
  transformation[0][1] = uxy*(dc1 - c) - uz*s;
  transformation[0][2] = uxz*(dc1 - c) + uy*s;
  transformation[1][0] = uxy*(dc1 - c) + uz*s;
  transformation[1][1] = uyy + (dc1 - uyy)*c;
  transformation[1][2] = uyz*(dc1 - c) - ux*s;
  transformation[2][0] = uxz*(dc1 - c) - uy*s;
  transformation[2][1] = uyz*(dc1 - c) + ux*s;
  transformation[2][2] = uzz + (dc1 - uzz)*c;

  GRX_ZERO(transformation[0]);
  GRX_ZERO(transformation[1]);
  GRX_ZERO(transformation[2]);
}
```

RAY-BOX PIERCE TEST C++ SOURCE CODE

Adapted from [23]

```
#define RIGHT 0
#define LEFT 1
#define MIDDLE 2

int
grx_raybox_pierce_test
(double minB[3], double maxB[3],  /* box */
 double orig[3], double dir[3],   /* ray */
 double coord[3])
{
  char inside = true;
  char quadrant[3];
  register int i;
  int whichPlane;
  double maxT[3];
  double candidatePlane[3];
  /* Find candidate planes; this loop can be avoided if
     rays cast all from the eye(assume perpsective view) */
  for (i=0; i<3; i++)
    if(orig[i] < minB[i]) {
      quadrant[i] = LEFT;
      candidatePlane[i] = minB[i];
      inside = false;
    }else if (orig[i] > maxB[i]) {
      quadrant[i] = RIGHT;
      candidatePlane[i] = maxB[i];
      inside = false;
    }else {
      quadrant[i] = MIDDLE;
    }
  /* Ray orig inside bounding box */
  if(inside) {
    coord = orig;
    return (2);
  }
  /* Calculate T distances to candidate planes */
  for (i = 0; i < 3; i++)
    if (quadrant[i] != MIDDLE && dir[i] !=0.)
      maxT[i] = (candidatePlane[i]-orig[i]) / dir[i];
    else
      maxT[i] = -1.;
  /* Get largest of the maxT's for final choice of intersection */
  whichPlane = 0;
  for (i = 1; i < 3; i++)
    if (maxT[whichPlane] < maxT[i])
      whichPlane = i;
  /* Check final candidate actually inside box */

  if (maxT[whichPlane] < 0.) return (0);
  for (i = 0; i < 3; i++)
    if (whichPlane != i) {
      coord[i] = orig[i] + maxT[whichPlane] *dir[i];
```

```
      if (coord[i] < minB[i] || coord[i] > maxB[i])
      return (0);
    } else {
      coord[i] = candidatePlane[i];
    }
  return (1); /* ray hits box */
}

#undef RIGHT
#undef LEFT
#undef MIDDLE
```

CLOSEST POINTS ON LINE SEGMENTS C++ SOURCE CODE

```
int
line_segment_intersect
(GRX_NODE *node0,
 GRX_NODE *node1,
 GRX_NODE *node2,
 GRX_NODE *node3,
 double *isect_pt,
 bool bounds_check_override)
{
  double orig0[3], orig1[2], dir0[3], dir1[3];
  double p2mp1[3], p2mp1xv2[3], LHS[3], RHS[3];
  double a, dx, dy, dz, length, length0, length1, max_length;
  double isect_pt0[3], isect_pt1[3];
  double zero_edge_length = 1.e-12; //1.e-6 ^2
  GRX_SET(orig0,node0->x_);
  GRX_SET(orig1,node2->x_);
  GRX_SUB(dir0,node1->x_,orig0);
  GRX_SUB(dir1,node3->x_,orig1);

  GRX_SUB(p2mp1,orig1,orig0);
  GRX_CROSS(p2mp1xv2,p2mp1,dir1);
  GRX_CROSS(LHS,dir0,dir1);
  GRX_CROSS(RHS,p2mp1,dir1);
  a = GRX_NORM(RHS) / GRX_NORM(LHS);
  if(GRX_DOT(LHS,RHS) < 0)
    a = -a;
  if(bounds_check_override) {
    GRX_TRAVEL(isect_pt,orig0,dir0,a);
    return 1;
  }
  if(a >= 0. && a <= 1.) {
    GRX_TRAVEL(isect_pt0,orig0,dir0,a);
    GRX_SUB(p2mp1,orig0,orig1);
    GRX_CROSS(p2mp1xv2,p2mp1,dir0);
    GRX_CROSS(LHS,dir1,dir0);
    GRX_CROSS(RHS,p2mp1,dir0);
    a = GRX_NORM(RHS) / GRX_NORM(LHS);
    if(GRX_DOT(LHS,RHS) < 0)
      a = -a;
    if(a >= 0. && a <= 1.) {
      GRX_TRAVEL(isect_pt,orig1,dir1,a);
      GRX_TRAVEL(isect_pt1,orig1,dir1,a);
      length0 = distance2_between_GRX_NODEs(node0,node1);
      length1 = distance2_between_GRX_NODEs(node2,node3);
      max_length = length0 < length1 ? length1 : length0;
      dx = isect_pt0[0] - isect_pt1[0];
      dy = isect_pt0[1] - isect_pt1[1];
      dz = isect_pt0[2] - isect_pt1[2];
      length = dx*dx + dy*dy + dz*dz;
      if(length / max_length > zero_edge_length) {
        return 0;
      }
    }
```

```
    else {
      return 0;
    }
  }
  else {
    return 0;
  }

  return 1;
}
```