EFFICIENT PARALLEL ALGORITHM FOR OVERLAYING SURFACE MESHES

A Thesis
Presented to
The Academic Faculty

by

Ankita Jain

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in
Computer Science

College Of Computing
Georgia Institute of Technology
August, 2007

EFFICIENT PARALLEL ALGORITHM FOR OVERLAYING SURFACE MESHES

Approved by:

Professor Xiangmin Jiao, Advisor
College Of Computing
Georgia Institute of Technology

Professor Jarek Rossignac
College Of Computing
Georgia Institute of Technology

Professor Hongyuan Zha
College Of Computing
Georgia Institute of Technology

Date Approved: May 8, 2007

To my parents Arun, Renu, my sister Samridhi, and my brother Vishesh

for their love, encouragement, and support.

ACKNOWLEDGEMENTS

I want to express my heartfelt gratitude to my advisor Dr. Xiangmin Jiao, without whose help and encouragement I would have never embarked upon this journey. When short of motivation, I always looked up to him for some and he never failed to inspire me with his passion for the subject and desire for perfection. I have also learnt from him serious scientific research attitude and the art of technical writing. I would also like to thank Dr. Jarek Rossignac for agreeing to serve on my committee. It was an honor for me to be his student and learn about different concepts in geometry from him. I also feel privileged to have Dr. Hongyuan Zha on my committee. I sincerely thank him for his advice and suggestions on the thesis. Last but not the least, I want to thank all my friends and family members for standing by me and having faith in me. I specifically want to thank my aunt Sangeeta, who has been like a mother to me. Finally, I want to thank my parents, Arun and Renu, and my sister, Samridhi, and brother, Vishesh, for their unconditional support and love.

TABLE OF CONTENTS

DEDICATION . . . iii

ACKNOWLEDGEMENTS . . . iv

LIST OF FIGURES . . . vii

SUMMARY . . . ix

I INTRODUCTION . . . 1

1.1 Terminologyy . . . 3

1.1.1 Surface Mesh . . . 4

1.1.2 Common Refinement . . . 4

II RELATED WORK . . . 6

III REVIEW OF THE ORIGINAL ALGORITHM . . . 11

3.1 Primitives . . . 11

3.1.1 Point Projection . . . 11

3.1.2 Edge Intersection . . . 13

3.2 Old Algorithm . . . 15

3.2.1 Phase One: Locating Subvertices . . . 15

3.2.2 Phase Two: Determining Subfaces . . . 17

IV MODIFIED ALGORITHMS AND PARALLELIZATION . . . 18

4.1 Higher Level Primitive . . . 18

4.1.1 Face Intersection . . . 18

4.1.2 Safeguarded Point Projection . . . 20

4.2 Simplified Serial Algorithm . . . 23

4.2.1 Finding image of blue mesh B on green mesh G . . . 24

4.2.2 Locating subvertices . . . 26

4.2.3 Resolution of inconsistencies . . . 31

4.2.4 Making subfaces and Triangulation . . . 32

4.2.5 Analysis of complexity . . . 34
4.3 Parallel Algorithm . . . 35
4.3.1 Step one - Discovering green mesh subsets . . . 36
4.3.2 Step 2 - Computing subvertices . . . 38
4.3.3 Step 3 - Resolving inconsistencies . . . 40
4.3.4 Step 4 - Making subfaces . . . 41
4.3.5 Step 5 - Redistributing subfaces . . . 41
4.4 Comparison of algorithms . . . 41
V RESULTS . . . 45
VI APPLICATIONS AND CONCLUSION . . . 53
6.1 Applications . . . 53
6.1.1 Multi-physics Systems . . . 53
6.1.2 Fluid-Solid Interaction . . . 54
6.1.3 Graphical Applications . . . 55
6.1.4 Biomedical Applications . . . 56
6.2 Conclusion . . . 57
REFERENCES . . . 60

LIST OF FIGURES

1 Two meshes (blue and green) modeling the same surface and their common refinement (black) . . . 3

2 Point projection from green (lower) to blue (upper) mesh . . . 12

3 Illustration of edge intersection . . . 15

4 Connecting each pair of subvertices that are on the boundary of same blue (solid) face and also on the same green edge . . . 16

5 Enumerating subfaces using divide-and-conquer. Dark subedges indicate current list of subvertices being processed, and arrows indicate traversal of subedges . . . 17

6 Projection of blue vertex to green edge . . . 19

7 Projection of blue vertex to green vertex . . . 19

8 Projection of a point to bilinear surface passing through normals of vertices of edges on the other mesh . . . 21

9 Inversion of normals while projecting points from blue mesh to green mesh . . . 23

10 Projection of blue mesh B on green mesh G . . . 26

11 Adding potential green faces that could intersect with a blue face . . . 28

12 Duplication of blue-green edge pair intersection . . . 29

13 Representation of a face . . . 30

14 Different types of inconsistencies and their remedies . . . 31

15 Subfaces using green and blue edge constraints. Blue arrows are blue constraints and green arrows are green constraints. Subfaces are made following the constraints of common subvertices between a blue and green face pair. . . . 33

16 Sorting subvertices common to a blue-green face pair to form subfaces 35

17 Mesh partitioning demonstrating that the blue and green partitions owned by a processor do not overlap . . . 37

18 Every processor makes bounding boxes around its blue mesh partition to communicate its spatial extent to the other processors . . . 38

19 Inconsistencies that can arise along the green partition boundaries. . . 39

20 Inconsistencies along blue partition boundary . . . 41

21 Speedup achieved for parallel code 46

22 Blue mesh (above) and green mesh (below) used for illustration. The meshes have been divided into four subdomains with different colors representing different subdomains. Red, green, blue and cyan represent subdomains 1, 2 ,3 and 4 respectively. 47

23 Input meshes (blue above, green below) and corresponding common refinement for processor 1 48

24 Input meshes (blue above, green below) and corresponding common refinement for processor 2 49

25 Input meshes (blue above, green below) and corresponding common refinement for processor 3 50

26 Input meshes (blue above, green below) and corresponding common refinement for processor 4 51

27 zoomed in view of common refinement 52

SUMMARY

Many computational applications involve multiple physical components, each having its own computational domain discretized by a mesh. An integrated simulation of these physical systems requires transferring data across the boundaries, which are typically represented by surface meshes composed of triangles or quadrilaterals and are non-matching with differing connectivities and geometry. It is necessary to construct a common refinement (or common tessellation) of the surface meshes to transfer data between different domains accurately and conservatively. For large-scale problems that involve moving boundary, the common tessellation must be updated frequently within the integrated simulations running on parallel computers.

Previously, Jiao and Heath developed an algorithm for constructing a common tessellation by overlaying the surface meshes. The original algorithm is efficient and robust, but unfortunately, it is complex and difficult to parallelize. In this thesis, we develop a modified algorithm for overlaying surface meshes. Our algorithm employs a high-level primitive, face-intersection, which combines the low-level point-projection and edge-intersection primitives of the original algorithm. A main advantage of our modified algorithm is its ease of implementation and parallelization. Our implementation utilizes flexible data structures for efficient computation and query of the common tessellation and avoids potential redundancy in computations to achieve high efficiency. To achieve robustness, we pay special attention to avoid potential topological inconsistencies due to numerical errors, and introduce a preprocessing step to project a far-apart surface mesh onto other before computing the common tessellation. We present numerical examples to demonstrate the robustness and efficiency of

our method on parallel computers.

CHAPTER I

INTRODUCTION

There are a wide range of scientific applications that consist of multiphysics systems. Such systems consists of subdomains exhibiting diferent physical behaviors, which often are mutually interacting. Examples include fluid-structure interactions as in foil flutter in airplane wings, thermal mechanical coupling such as in sections of bridges and highways, bio-physics, automotive industry, materials science, etc. Simulation of such systems is a great challenge in the computational science area, because these physical processes are represented by sophisticated models and methods, and there is a need for more elaborate methods for their integration and interaction. Such interactions are very convoluted because of the disparity in the representations of the different sub-systems involved.

There are many factors that affect the accuracy of the simulation, including proper physics models, accurate spatial discretization, and efficient communication and data transfer between different components. The different systems involved in the simulation have their own computational domains, which are discretized into meshes. Such volume meshes consist of polyhedras, such as tetrahedras or hexahedras. The different sub-systems involved communicate at the boundaries with other sub-systems. So the boundaries or interfaces between these components have more than one realization, one for each subdomain abutting the interface. An integrated simulation of the entire system involves data transfer at these common boundaries and interfaces. These boundaries are represented by surface meshes composed of polygons like triangles and quadrilaterals. The treatment of interaction at these boundaries involves a lot of issues. For example: do the meshes match? Can meshes slip past each other?

These surface meshes are in general non-matching with differing connectivities and geometry and in large scale problems these meshes often involve moving boundaries. Finding a method for data transfer that is both numerically accurate and physically conservative becomes a non-trivial task.

The subject of this thesis is the computation of a data-structure that helps and expedites the process of transferring data between differing mesh representations of a surface model that represent the interfaces in many scientific application. These systems have extensive computational demands and need successful utilization of high performance computing systems which demands research and progress in finding algorithms and methods that lend themselves to parallelization.

In [8], Jiao and Heath introduced a data structure called the common refinement of two meshes, which is crucial for such data transfers. They developed a method to exchange data across the interface meshes in [7], which was both accurate and conservative, utilizing the common refinement. A common refinement of two meshes is a mesh composed of polygons that subdivide the polygons of both input meshes simultaneously. It defines, and allows efficient query of, a unique nearby corresponding point on one surface for every point on the other. Albeit the original algorithm was very efficient and robust, it was difficult to implement and parallelize. So the main aim of this thesis is to modify the original algorithm to counter its complexities involved in the implementation and its inherently sequential nature.

Our first contribution, in this thesis, is the simplification of the serial implementation of computing common refinement while maintaining the accuracy and robustness of the original algorithm. The numerical errors, caused by the ineaxct computation of the non-linear primitives used in the algorithm, can introduce potential numerical and topological inconsistencies. The algorithm employs techniques for detection and automatic resolution of such inconsistencies in order to increase the robustness of the algorithm. We introduce a higher-level primitive, face-face intersection, which in

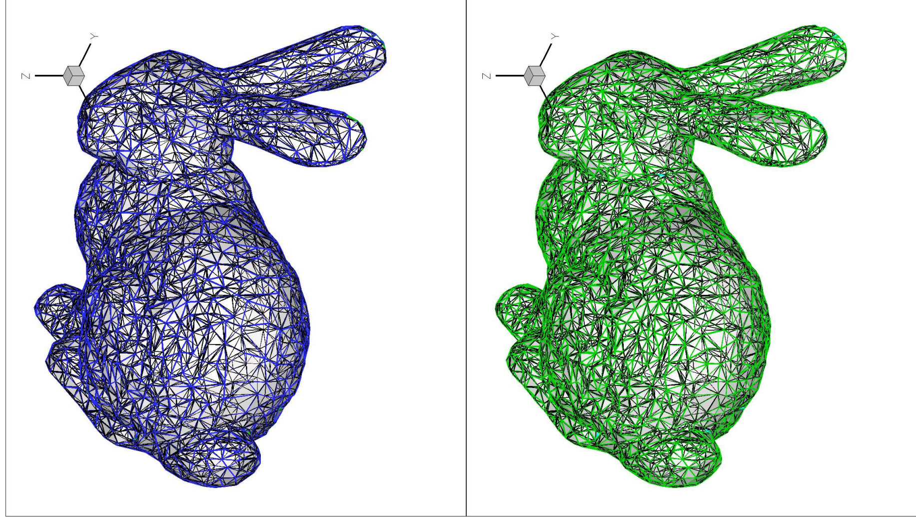


Figure 1: Two meshes (blue and green) modeling the same surface and their common refinement (black)

addition to simplifying the whole process, also streamlines the logic for analyzing the inconsistencies. Our second contribution is the introduction of a new preprocessing step which improves the robustness of the algorithm. The new step involves finding a new mesh (lets say B' if we assume that one of the input meshes is B), which is the image (projection) of the old input mesh B on the input mesh G and has the same connectivity and geometry of the old mesh B. Using B' in the algorithm in place of the original mesh, helps in the cases when two surfaces are far apart and leads to more robust and accurate results. Our third contribution is the support of partially overlapping meshes. Also, for meshes that have disparate topologies, the new algorithm can build correspondence whenever possible, unlike the original algorithm which required the input meshes to have same topology and matching features.

1.1 Terminologyy

This section explains some of the terms that will be used throughout this thesis.

1.1.1 Surface Mesh

A surface mesh is a discretization of the geometry of a surface. It a tessellation of a surface into polygons. Topological objects of any dimension in a mesh are called *cells*. We refer to the 0-dimensional cells as *vertices*, the 1-dimensional cells as *edges*, and the 2-dimensional cells as *facets*. A surface mesh is then defined as collections of these cells embedded in $\mathbb{R}^3$. Vertices are connected to form edges. The edges are then joined to form facets. Each cell σ has a geometric realization in $\mathbb{R}^3$, which is the point set $\{\Sigma_i N_i x_i \mid \Sigma_i N_i = 1 \, \text{and} \, N_i \geq 0\}$, where the x_i are the vertices of the cell, and the $N_i(\varepsilon, \eta) : \mathbb{R}^3 \rightarrow \mathbb{R}$ are the associated shape functions. The line segment between the two vertices is the realization of an edge, and that of a triangle or quadrilateral is a linear or a bilinear patch bounded by their edges.

1.1.2 Common Refinement

A mapping $f : X \rightarrow Y$ is continuous if the preimage or every open set in Y is an open set in X. Two sets X and Y are *homeomorphic*, denoted by $X \approx Y$ if there is a bijective and continuous mapping $f : X \rightarrow Y$ whose inverse is also continuous; such an f is called *homeomorphism.* A mesh R is a *refinement* of a mesh M if their geometric realizations $\mid R \mid$ and $\mid M \mid$ are homeomorphic and every cell of M is partitioned into one or more cells of R. Given two homeomorphic meshes, a *common refinement* of them is a mesh that is a refinement of both given meshes.

A mesh R is called refinement of mesh M if R subdivides the polygons of M into one or more polygons and their geometric realizations are homeomorphic (i.e. there is a structure-preserving map between the two). Common refinement of two homeomorphic meshes then is a mesh M which subdivides the facets of both the input meshes simultaneously.

Therefore, common refinement of two meshes is a combinatorial union of them. It subdivides the polygons of both the input meshes simultaneously, with each mesh

containing one of the two realizations of every cell of the common refinement. It provides a correspondence between a point on one mesh and a nearby unique point on the other mesh.

The cells of the common refinement R are referred to as *subcells*. In particular we refer to the 0-, 1-, and 2-dimensional subcells as *subvertices*, *subedges* and *subfacets,* respectively. A subvertex is either a blue or green vertex, or an intersection of a blue and a green edge. The subedges of the overlay are the intervals in blue and green edges between the subvertices. Cells in blue mesh B and green mesh G, host some cells in the common refinement. The lowest dimensional blue or green cell that contains a cell in the common refinement is called the blue and green parent, respectively. Hence, every subcell of a common refinement has two geometric realizations, one contained in the realization of either of its parents. We say that a common refinement R is *minimal* if no two subcells in R share the same blue and green parent, which implies that the size of the output cannot be reduced by merging the subcells without changing their parents.

CHAPTER II

RELATED WORK

Pervasiveness of multi-physics systems has led many researchers to be interested in finding mappings between meshes modeling the same interface but having different combinatorial structures. Mesh correspondence is also an important step in many graphical techniques, such as morphing.

There are two ways specifying correspondence between different meshes:

- *Mesh Association*: Pointwise interpolation techniques, used to transfer data between non-matching meshes, require vertices of one mesh to be associated with the facets of the other mesh. This association then used to find local coordinates of the vertices of one mesh in the other mesh. This approach of associating vertices in one mesh with facets in the other mesh is called *mesh association*.

- *Mesh Overlay*: Mesh overlay refers to the technique of computing a mesh that subdivides the polygons of both the input meshes simultaneously. Such a mesh is referred to as the common refinement of two meshes. It defines and allows efficient query of a unique nearby corresponding point on one surface for every point on the other.

There are various algorithms to construct mesh association and overlay, each with its own advantages and disadvantages. There are some very simple algorithms like the nearest-neighbor method and normal-projection method. The nearest neighbor method finds for every point in one mesh a corresponding point in other mesh, which has the least distance from it. Such an association is not one-to-one. The normal

vector association defines the image of a point x in mesh A as its projection onto mesh B along the normal directions to A at x. The mappings found with this method are not unique because the normal vector is not continuous along features. Also, the mappings found with these two methods are only injective and not necessarily surjective. For instance, along the boundaries there can be points in B which do not have any corresponding point in A.

Some methods try to find projection directions, which are continuous so that they can be used for finding one-to-one mapping between two meshes. One such method has been proposed by E.H. Van Brummen in [17]. The paper proposed a method that defines the image of a point on a mesh as its projection along the so-called normal vector field onto the other mesh. The smooth normal vector field is defined by the solution of a modified Helmholtz equation with right hand side data corresponding to the normal-vector field. The increase in the smoothness of the geometry representations is well utilized by their algorithm.

Some mesh association methods use a virtual interface surface onto which the points of both the meshes are projected. These methods then use the fact that the resulting virtual surface now contains the topology of both the input models, making it easier to find correspondence between them. Alexa has described one such method in [1]. It involves merging two genus 0 polyhedra that establishes correspondences between vertices of the models. This is achieved by first embedding the surface on a unit spheres, tweaking the embeddings to match features on the two models, and then computing an overlay of the meshes on the sphere. The main drawback of using virtual surfaces is that only the models that have topologically equivalent embedding on the virtual surface can be used. It also does not guarantee a bijective mapping. Beckert has presented a coupling scheme in [2] using finite interpolation elements to couple fluid (CFD) and structural models (FE) for aeroelasticity, which involves transforming the aerodynamic loads into respective work equivalent forces at the

FE nodes and calculating the displacements at the aerodynamic surface mesh using displacements and rotations of the surface model. Another scheme to couple fluid and structure has been presented by Quaranta et al in [14]. Based on the moving least square (MLS) patches technique used in the field of surface reconstruction from N irregularly distributed given data points.

Establishing homeomorphism between surfaces is an integral operation of many graphical applications. Chazal et al. [3] defined a mapping (Orthomap) between two (n-1)-dimensional manifolds S and S' in R^n, which associates with point x on S with the closest point on S' lying on the line passing through the normal to S at x. This mapping is homeomorphic if the minimum feature sizes of S and S' both exceed $h/(2-\sqrt{2})$, where h is the Hausdorff distance between S and S'. Chazal et al. [4] proposed a ball mapping that relaxed the the minimum feature sizes of the two surfaces to exceed h, for a homeomorphic mapping. Both these schemes can scale up to (n-1)-dimensional manifolds, for any n. Our algorithm in contrast on meshes modeling a 2-manifold.

Schreiner et al. proposed another method in [15] to establish an association between two meshes. They constructed a continuous bijective mapping between two triangle meshes M^1 and M^2. The method involved partitioning the two surfaces into a corresponding set of triangular patches by tracing a set of corresponding paths and then creating progressive mesh representations of both input meshes M^1 and M^2, starting with two base meshes with identical connectivities. A trivial map is first defined between these base meshes, and then the map is iteratively refined resulting in an inter-surface map. The map produced here is very similar to the one produced by the common refinement, in the sense that the linear pieces of the map are finer then the original mesh faces. The meta-mesh formed is the union of the input meshes with vertices of the meta mesh including vertices of both the original meshes as well as vertices formed by edges of M^1 intersecting those of M^2. They also specify for each

mesh vertex, the face of the other mesh to which it maps along with the barycentric coordinates within that face and for each edge-edge intersection, the two ratios formed by the split point on each edge. Further the polygonal regions are triangulated. In [16], Turk and Levoy have proposed a zippering scheme to combine a collection of range images into a single polygonal mesh that completely describes an object. They used an iterated closest point algorithm to find correspondance between two different meshes.

Another algorithm which parameterizes the models over a common base mesh domain was proposed by Kraevoy and Sheffer in [10]. They constructed a common base mesh domain from topologically identical triangular layouts of the two meshes, which are then mapped to the corresponding base mesh triangles.This helps in the computation of the initial mapping which is then refined.

An alorithm to compute overlay of planar convex subdivisions was proposed by Guibas and Seidel in [6]. The have devised a topological line sweep algorithm to find the overlay.

The systems comprising of myriad complex interacting physical phenomena make use of meshes that can have elements in order of millions. These meshes are generally partitioned across several processors to exploit the benefits of parallel computing. The methods mentioned above are advantageous and produce good results in certain situations but they have not been extended to handle partitioned meshes. Most multi-physics systems require significant amount of resources like memory, CPU time etc. Advances in parallel computing can be used to alleviate such situations.

The biggest challenge that is faced in parallelizing the mesh overlay algorithms is due to the fact that the input meshes are partitioned and distributed across different processors independently of each other. This creates the need for every procesor to be aware of the partitions lying on the other processors. We cannot assume that the mesh partition information is globally known. So a processor has no knowledge of which

processor owns every mesh element and has to communicate with other processors to generate this information. Interprocessor communication costs in parallel algorithms can greatly degrade the system performance and diminish the potential benefits of utilizing increased number of processors.

In [13], Plimpton et al. discussed such a situation and have put forward a method to overcome it. They use a rendezvous technique wherein a third decomposition is used to search for the elements in one grid that contains the nodal points of the other. They used Recursive Coordinate Bisection (RCB), a geometric partitioning algorithm, which assigns a geometric region to a rendezvous processor. The algorithm then requires every processor to send the corresponding portions of both the input meshes to the rendezvous processors.

Farhat et al in [11] proposed a parallel algorithm to find association between non-confirming fluid/structure interfaces. The algorithm proposed provided a matching between two meshes at the interface and achieves good load balancing.

CHAPTER III

REVIEW OF THE ORIGINAL ALGORITHM

This chapter explains the original algorithm from [8]. It has two phases. In the first phase, all the sub-vertices of the overlay and their blue and green parents are identified and then the subvertices are sorted in their host edges. In the second phase, the parent information is used to determine subfacets of the overlay.

3.1 Primitives

3.1.1 Point Projection

Two types of projections are commonly used to correlate points on two surfaces, namely orthogonal projection and closest-point projection. These projections map a point on one surface onto a nearby point on another surface, and vice versa, which is reasonable for some applications. Unfortunately, neither of these projections can possibly define a homeomorphism between two mesh surfaces, because a triangular or quadrilateral mesh surface is in general not smooth at edges or vertices. We construct a nearly orthogonal projection by interpolation using the shape functions of the green mesh. We first define the projection from $\mid G \mid$ to $\mid B \mid$, and will use its inverse to define the projection from $\mid B \mid$ to $\mid G \mid$. For any point $\mathbf{p} \in \mid G \mid$, we can write its projection $\mathbf{q} \in \mid B \mid$ as $\mathbf{p} + \gamma(\mathbf{p}) d(\mathbf{p})$, where $\mathbf{d} : \mid G \mid \rightarrow \mathbb{S}^2$ denotes the normalization of $\mathbf{q} - \mathbf{p}$, and $\gamma : \mid G \mid \rightarrow \mathbb{R}$ denotes the length of $\mathbf{q} - \mathbf{p}$. We construct such a function d by first fixing its values at the green vertices, which can be given by the user or evaluated numerically as the average of the facet normals.

Let $\mathbf{x_i}$ denote the ith vertex, and $\mathbf{d_i}$ denote its associated unit vector. For a point $\mathrm{p} = \mathbf{p} = \Sigma_{\mathrm{i}} N_{\mathrm{i}} \mathbf{x_i} \in \mid G \mid$, its direction $\mathbf{d}$ is then the normalized vector of $\Sigma_{\mathrm{i}} N_{\mathrm{i}} \mathbf{d_i}$ as illustrated in Figure 2. The continuity of d follows from the continuity of the N_i.

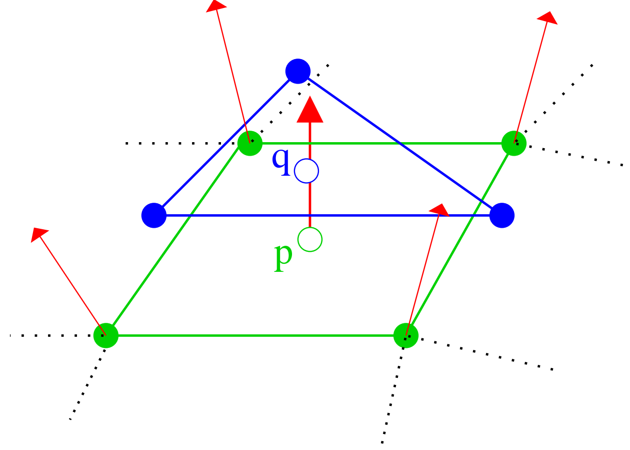


Figure 2: Point projection from green (lower) to blue (upper) mesh

The main virtues of this projection are that it is close to orthogonal for reasonably smooth surfaces, its projection directions form a continuous vector field over the surface, and it is convenient to compute. Approximate orthogonality is desirable for accurate data transfer, the main application of common refinement. It is also important for good conditioning of the equations that must be solved for the primitives. The continuity of $\mathbf{d}$ makes it possible for the point correspondence to be a homeomorphism. We note that for closed surfaces if the projection is nearly orthogonal, the continuity of $\mathbf{d}$ implies that there exists an ϵ that depends on the smoothness of $\mathbf{d}$, such that the point correspondence is a homeomorphism if the Hausdorff distance (i.e., the max-min distances between two point sets) between the surfaces is smaller than ϵ. For now, we shall assume the two meshes are modeling a closed smooth surface, and they are close to each other so that the projection is a homeomorphism. These assumptions become invalid for surfaces with ridges, corners, or nonmatching boundaries. Note that when the projection is a homeomorphism, its inverse exists, which gives the projection from $\mid B \mid$ to $\mid G \mid$.

This primitive is defined piecewise on the facets of the green mesh, and it is very convenient to evaluate. In particular, the projection of a point $\mathbf{p} \in \mid G \mid$ onto a facet

$b \in B$ is the intersection of b with the line passing through $\mathbf{p}$ with direction $\mathbf{d}(\mathbf{p})$. If $\Sigma N_{\mathrm{i}} \mathbf{x_i} \in b$ is the projection of $\mathbf{p}$, then we can pose the problem as a system of equations

$$\Sigma N_i \mathbf{x_i} - \mathbf{p} - \gamma \mathbf{d} = 0, \tag{1}$$

Because N is a function of the two local coordinates of b, this system has three equations (one for each component of the physical coordinates) and three unknowns (γ and the local coordinates). The system is linear if the blue facet b is a triangle but is bilinear if b is a quadrilateral. In the latter case, we solve the system using Newton's method, which converges quadratically when $\mathbf{d}$ is nearly orthogonal to the facet. The primitive returns the local coordinates of the projection and reports whether $\mathbf{p}$ projects to the interior, an edge, a vertex, or the exterior of the facet, identified from the barycentric coordinates.

The projection from a point $\mathbf{q} \in \mid B \mid$to a green facet $g \in G$ can be computed similzirly. Let the point $\Sigma N_i \mathbf{x_i}$ be its projection in g whose projection direction is $\Sigma N_i \mathbf{d_i}$. We then have the system of nonlinear equations

$$\Sigma N_i \mathbf{x_i} - \mathbf{q} - \Sigma N_i \mathbf{d_i} = 0, \tag{2}$$

which also has three unknowns and can be solved similarly.

3.1.2 Edge Intersection

Another primitive for mesh overlay is to compute the intersection of a blue edge b $\in$ B with a green edge g $\in$ G. Again, the intersection has two realizations, one in $|B|$ and one in $|G|$. This primitive determines both realizations and reports whether the intersection is in the interior, at a vertex, or the empty set. Let the blue edge be b $=$ $\mathrm{b}_0\mathrm{b}_1$ and the green edge g $=$ g_0 g_1, and let the projection directions at g0 and g1 be

d0 and d1 respectively. We parameterize b by $b_0 + \alpha(b_1 - b_0)$, and parameterize g by $g_0+\beta(g_1-g_0)$. The projection direction of a point in g is then given by $d_0 + \beta(d_1 - d_0)$, and hence

$$b_0 + \alpha(b_1 - b_0) = g_0 + \beta(g_1 - g_0) + \gamma(d_0 + \beta(d_1 - d_0)), \tag{3}$$

which again has three equations and three unknowns. This formulation, unfortunately, is very ill-conditioned when b and g are nearly parallel, which happens frequently in practice, and Newton's method can converge very slowly or fail to converge with this formulation. We hence reformulate the equation as follows. Note that the realization of the intersection point in g is in the plane passing through b with normal direction $n = (b_1 - b_0) \times (d_0 \times \beta(d_1 - d_0))$, as illustrated in Figure 3. Therefore, we have the equation

$$n \cdot (g_0 + \beta(g_1 - g_0) - b_0) = 0 \tag{4}$$

Substituting n and reordering the equation leads to a quadratic equation $c_2\beta^2 + c_1\beta + c_0 = 0$, where

$c_2 = ((b_1 - b_0) \times (d_1 - d_0)) \cdot (g_1 - g_0),$

$c_1 = ((b_1 - b_0) \times d_0) \cdot (g_1 - g_0) + ((b_1 - b_0) \times (d_1 - d_0)) \cdot (g_0 - b_0),$

$c_0 = ((b_1 - b_0) \times d_0) \cdot (g_0 - b_0)$

which can be solved analytically. The realization in b is the intersection of the edge b with the plane passing through g with normal $l = (g_1 - g_0) \times (d_0 \times \beta(d_1 - d_0))$. Therefore, the parameter α is the solution to the linear equation

$$l \cdot (b_0 + \alpha(b_1 - b_0) - g_0) = 0,$$

which can be solved after we have obtained β, provided $l \cdot (b_1 - b_0) \neq 0$. The intersection is in the interior of b if $\alpha \in (0, 1)$, at a vertex if $\alpha = 0$ or 1, in the exterior

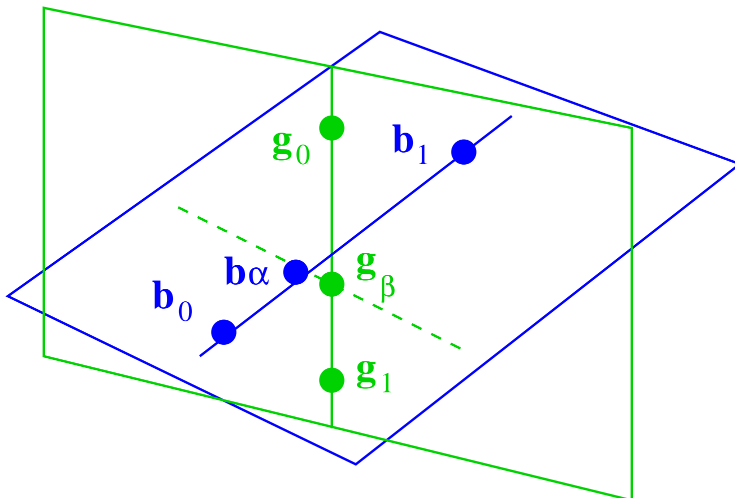


Figure 3: Illustration of edge intersection

otherwise; similarly for β. A solution corresponds to an actual edge intersection if $\alpha \in [0,1]$ and $\beta \in [0,1]$ simultaneously.

3.2 Old Algorithm

3.2.1 Phase One: Locating Subvertices

Subvertices are located in the following three steps:

1. locate subvertices along blue edges

2. sort subvertices in their green host edges

3. determine blue parents for remaining green vertices

3.2.1.1 Locating subvertices along blue edges

We traverse B in a breadth-first order and for each edge $b \in B$ locate the subvertices in b from one end to the other. Suppose b is a blue half edge and r_0 is the subvertex corresponding to $org\ b$, whose green parent is known. We find the other subvertices $r_1, \ldots, r_m, r_{m+1} = dst\, b$ in edge b and determine their parents, where $r_1, \ldots, r_m$ are in the interior of b sorted from $org\ b$ to $dst\ b$. We locate r_i in increasing order of i and intersect b with the green cells that can possibly host r_i . This operation is repeated

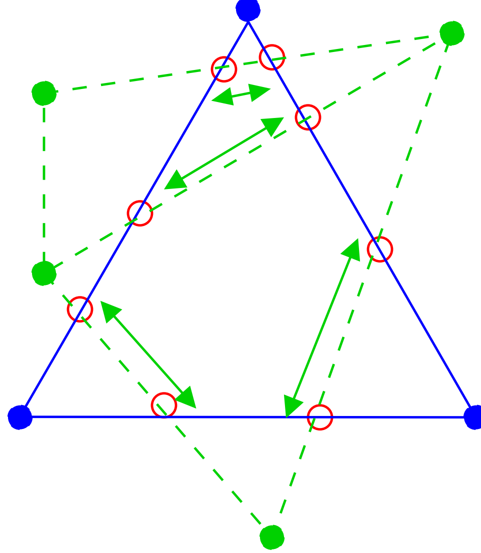


Figure 4: Connecting each pair of subvertices that are on the boundary of same blue (solid) face and also on the same green edge

until all blue edges have been processed. To start the algorithm, it is necessary to know the green parent of some blue vertex, which can be obtained by brute force.

3.2.1.2 Sort subvertices in their green host edges

Two adjacent subvertices in a green edge must be contained in a common blue facet. Therefore, blue facets are traversed to identify the adjacency of subvertices in green edges. Specifically, for each blue facet, the subvertices are grouped in its edges and vertices based on their green parents, with a green edge containing at most two subvertices in its group. If a group has two subvertices, the one that is closer to the origin of the green edge is identified as the predecessor of the one that is farther, and they are linked with each other, as illustrated in Figure 4

3.2.1.3 Determine blue parents for remaining green vertices

We still must determine the blue parents for the green vertices that are not in a blue edge or a blue vertex. This step can be considered the dual of step 1.

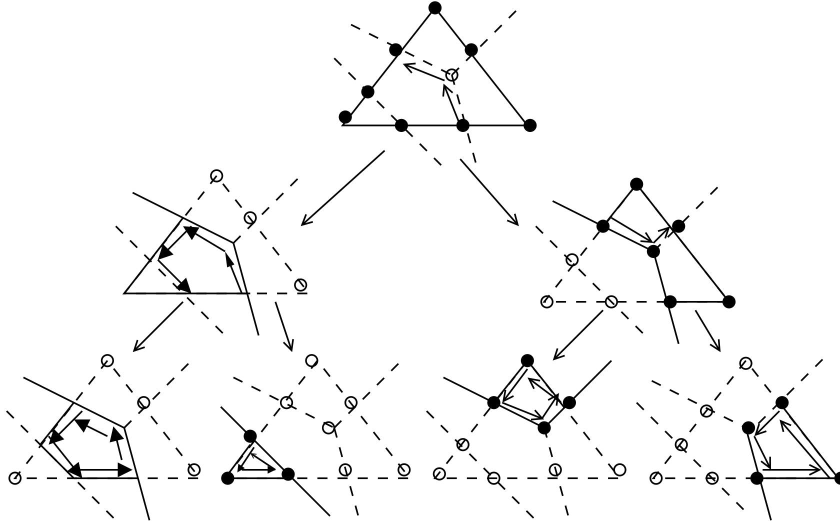


Figure 5: Enumerating subfaces using divide-and-conquer. Dark subedges indicate current list of subvertices being processed, and arrows indicate traversal of subedges

3.2.2 Phase Two: Determining Subfaces

This phase enumerates the subfacets contained in a given blue or green facet. The basic idea of the procedure used in this phase is to use divide-and-conquer as illustrated in Figure 5. First, a list of the subvertices in the edges of the blue facet in counterclockwise order, is created. Given a list, two adjacent subvertices in the list are taken, and their subedge is walked along and a left turn is made whenever a subvertex is reached, until all subvertices in the list have been visited or the visited subvertices cut the list into two parts. In the former case, the list of subvertices determines a subfacet. In the latter case, the original list is split into two and call the procedure is recursively called on each of them.

CHAPTER IV

MODIFIED ALGORITHMS AND PARALLELIZATION

4.1 Higher Level Primitive

4.1.1 Face Intersection

The original algorithm traversed the blue mesh B edge-by-edge to find subvertices. Even though edge traversal is more efficient for finding the subvertices, in the new algorithm we traverse the blue mesh B face-by-face because it makes the algorithm easier to implement and parallelize. To support this traversal we introduce a new primitive, face intersection, which is based on the two primitives, point projection and edge intersection. It is applied to a pair of green and blue faces while trying to find the common subvertices hosted by them. The vertices of the blue face are projected on the green face using (2). Now using the relative position of the blue vertices with respect to the green face, the edge intersection primitive is only applied to those pairs of green and blue edges that could potentially intersect.

As shown in the Figure 6 the blue vertex b_{v0} projected on the green edge g_{e0} when it was projected on the green face using the point projection primitive. Now we have the intersection of the blue edge b_{e0} and the green edge g_{e1}, and also the intersection of blue edge b_{e2} and the green edge g_{e1}. So now we would not apply the edge intersection primitive to the edge pairs (b_{e0}, g_{e1}) and (b_{e2}, g_{e1}).

In Figure 7 the blue vertex b_0 is projected onto the green vertex g_1. Now we have the intersection of the two edges incident on the blue vertex b_0 and the two edges incident on green vertex g_1, so we don't have to apply the edge intersection primitive to the edge pairs (b_{e0}, g_{e0}), (b_{e0}, g_{e1}), (b_{e2}, g_{e0}) and (b_{e2}, g_{e1}) .

After finding the projections of all the blue vertices, we find the projection of

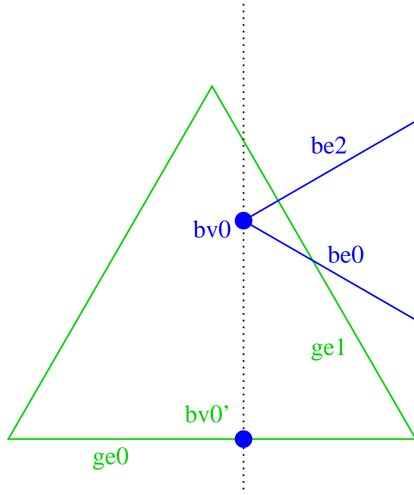


Figure 6: Projection of blue vertex to green edge

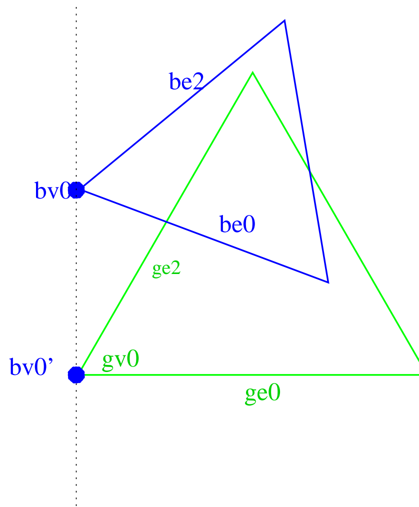


Figure 7: Projection of blue vertex to green vertex

all the green vertices and mark more pairs of blue and green edges for which the edge intersection primitive need not be computed for that pair of blue and green faces. After this we apply the edge intersection primitive to all the pairs of blue and green edges that were not marked above. In this way we combine the two lower level primitives, point projections and edge intersection primitives, to form the higher level primitive, face intersection.

4.1.2 Safeguarded Point Projection

In addition to all the primitives explained above, there is one more primitive for the new algorithm, which is used along with the point rejection primitive to find accurate projections of the points in the two meshes . The point projection primitive above tries to construct a nearly orthogonal projection by interpolation using the shape functions of the green mesh. But this primitive does not guarantee a right projection at features. The projection directions used to find these projections could be skewed near the features (ridges, sharp corners) or at the places with large curvature. This leads to wrong results. The normals at the vertices might get distorted and could cause the projections to be faulty too. Because of these skewed normal directions, a vertex can project inside more than one face at a time. This calls for a more restrictive condition to decide when a vertex from one mesh falls or projects inside a face on the other mesh. The new primitive tries to find whether a particular point on one mesh lies inside the area of influence of a face on the other mesh. Suppose we are trying to project a point P_b on mesh B onto face F_g of mesh G. The new primitive checks whether the point P_b would lie inside the volume bounded by the bilinear surfaces passing through the edges (in the direction of normals passing through the vertices of the edges) of the face F_g.

A bilinear surface is formed passing through the edge of face F_g in the direction of the normals at the two end points of the edge. If the edges of the face are considered to

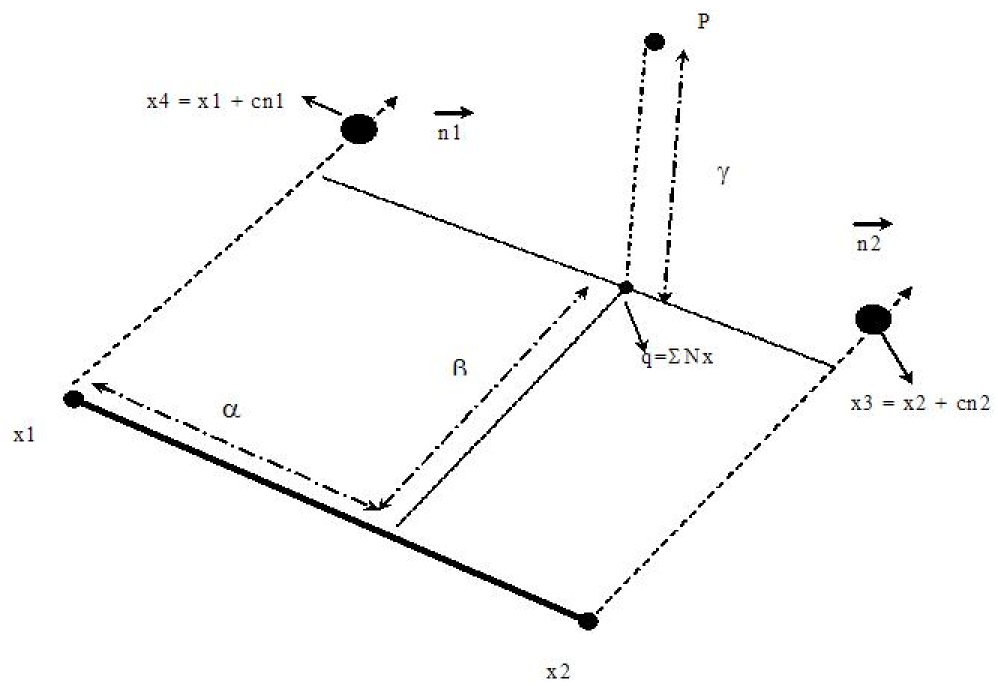


Figure 8: Projection of a point to bilinear surface passing through normals of vertices of edges on the other mesh

be in counter-clockwise order, then the point should lie to the left of bilinear surfaces passing through all the edges of the face. To find whether a particular point is to the left or right of the surface, it is projected on that surface and the sign of the distance from the point to the surface is used to determine which side of the surface the point lies. It is easy to see that the projection of the point P_b on the bilinear surface passing through an edge (in the direction of the normals at the end points of the edge) of the face is the intersection of the bilinear surface with the line passing through P_b in the direction perpendicular to the bilinear surface. This problem can be represented as a system of equations

$$\Sigma N_i x_i + \gamma d - P_b = 0, \tag{5}$$

where $\Sigma N_i x_i$ is the projection of the point P_b onto the bilinear surface and N_i are the associated shape functions or barycentric coordinates such that

$N_1 = (1 - \alpha)(1 - \beta)$

$N_2 = \alpha(1 - \beta)$

$N_3 = \alpha\beta$

$N_4 = (1 - \alpha)\beta$

with α, β are as shown in the Figure 8 and d is the direction of line passing through the point P_b and perpendicular to the bilinear surface such that

$$d = (x_2 - x_1 + c\beta(n_2 - n_1)) \times (\alpha n_2 + (1 - \alpha)n_1)$$

where the first vector of the cross product represents a vector in the direction of the bilinear surface and the second vector is in the direction of the edge. Hence their cross product gives a vector perpendicular to both the edge and the surface and thus is the direction of the projection from the point to the bilinear surface. γ is the projection distance, i.e., distance of point P_b to the bilinear surface along the direction d. We use the sign of γ to decide which side of the bilinear surface the point lies on. If this

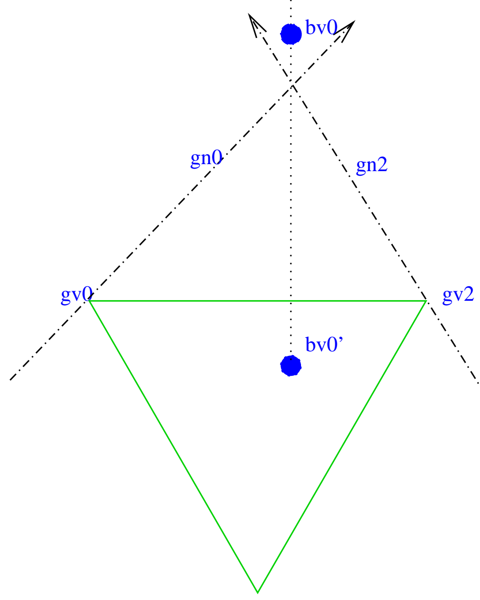


Figure 9: Inversion of normals while projecting points from blue mesh to green mesh

value is negative for all the edges, then the point lies inside the volume bounded by the three bilinear surfaces and it can project inside face F_g. The value of c should be chosen such that the quadrilateral $x_1x_2x_3x_4$ is not degraded or inverted i.e. no folding occurs in normals n_1 and n_2 as shown in Figure 9

We guarantee this by checking

$$\left[x_2 - x_1 + c\beta(n_2 - n_1)^T(x_2 - x_1) > 0\right].$$

If this condition is not satisfied then the point P lies outside the sphere of influence of the face in the other mesh and we ignore it. Also using this condition we find an upper limit for the value of β.

4.2 Simplified Serial Algorithm

The original algorithm in [8] was inherently sequential due to the locality-based search. This section proposes a modified version of the above algorithm which is easy to parallelize.

The algorithm can be laid down in four steps:

1. Preprocessing step: Finding the image (projection) of points of blue mesh on the green mesh
2. Locating subvertices
3. Forming subfaces (Triangulating)
4. Finding and correcting the inconsistencies

The following section explains all the steps in detail.

4.2.1 Finding image of blue mesh B on green mesh G

Procedure

The common refinement algorithm can be applied to any two surface meshes. For more accurate results, the algorithm assumes that the two meshes are close to each other in space. The point projection primitive explained above is close to orthogonal for reasonably smooth surfaces lying close to each other, but it might give inaccurate projections if the two meshes are far from each other. Also, the meshes need to be close to each other for any meaningful data transfer and to form well conditioned equations that have to be solved in the subsequent steps of the algorithm. In practice, the two meshes discretize the same interface, so they are congruous and generally close to each other in space.

For better results we project one mesh onto the other before computing the common refinement. This preprocessing step makes the algorithm more robust and also avoids folding when computing the subvertices. The aforementioned point projection primitive is used to project blue mesh B onto green mesh G in this step. Also, the primitive, safe guarded point projection, is used to check whether the projection found using point projection primitive is valid or not. All the points on the blue mesh B are projected onto the green mesh G, in other words, for every point b on

blue mesh we find a new point b' on the green mesh where b' is the projection of the point b on the green mesh G. A new mesh B' is formed from the projection of all the blue points by connecting them using the connectivity of the mesh B. Therefore, the meshes B and B' have same combinatorial structure. In essence, B' is the image of B on G. Now for all the subsequent steps in the computation of common refinement we will use the mesh B'. For every blue point in mesh B we store the green face it projects onto and the barycentric coordinates of that blue point with respect to that green face. The blue vertices that project outside the green mesh are tagged.

Implementation

We iterate the blue mesh face by face and try finding the projection of the vertices of the blue face on green mesh. To find projection of the vertices of the blue face we need to find a list of all the green faces onto which the blue vertices could potentially project. We use a data structure called kd-tree for finding the list. Kd-tree is a multidimensional search tree for points in k-dimensional space. We used 3D kd-trees. A kd-tree of centroids of all the faces in the green mesh is created. Then we make a bounding box around that blue face for which we want to determine which green faces its vertices can project onto. When given this bounding box as an input the kd-tree returns the centroids of the green faces that lie in that box. So our list of potential green face candidates becomes all the green faces whose centroids are in this list returned by the kdtree search operation. Basically, the kd-tree helps us to determine which green faces lie near the blue face and thus can hold the projection of the vertices of that blue face. One of the advantages of not integrating this step with actual subvertex location is that when we have moving (dynamic) meshes, this step can be ignored after the first pass and we can approximate the projection of blue points from their initial projections in the later passes. Kdtree is an expensive data structure in terms of the time it takes to build and search it. Separating this step

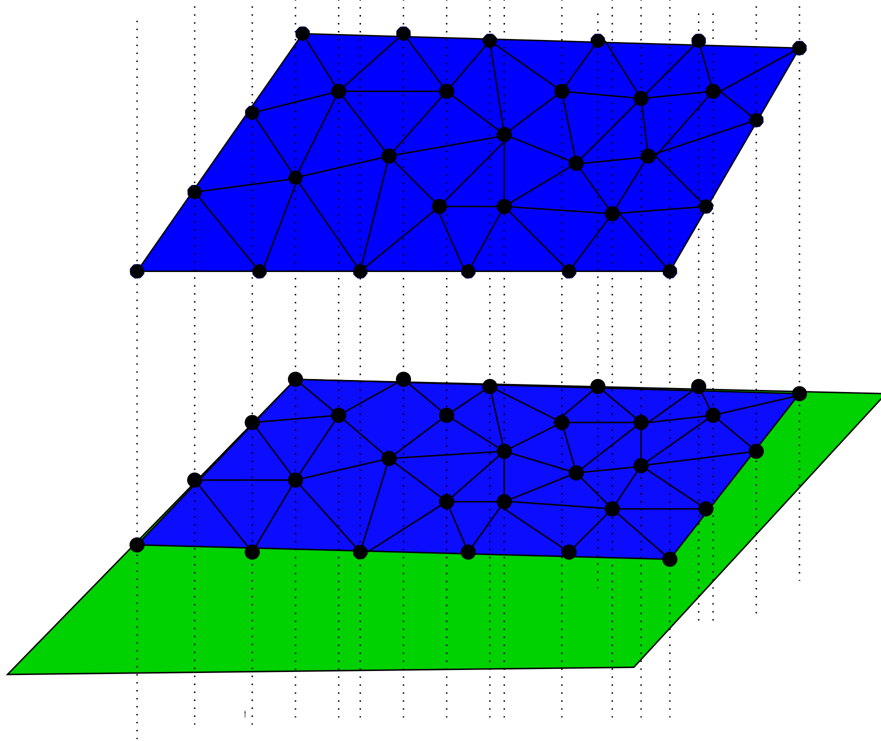


Figure 10: Projection of blue mesh B on green mesh G

from subsequent steps also helps keep the dependence of the computation process on kdtree data structure to the minimum and can be replaced by a faster method later. We have to find all the potential green faces that the blue face could intersect with. This module takes as input the blue mesh B and green mesh G and returns a list of projection of the blue points on green mesh.

4.2.2 Locating subvertices

This step of the algorithm locates the subvertices and also their blue and green parents. A subvertex is either a vertex in one of the meshes or is an intersection of a blue and green edge. Every subvertex has at least one green parent and one blue parent. This phase of the algorithm involves locating subvertices and their respective parents in the two edges. We iterate the blue mesh face-by-face and locate subvertices lying on the edges of the blue face, simultaneously storing their green parents. The algorithm picks a blue face, finds a list of green faces that could potentially intersect

with it, find intersections with blue face using face intersection primitive and store the subvertices found. This is accomplished through the following steps:

For every blue face:

1. Extract the projection of the blue vertices found in the previous step
2. Find the potential green faces that intersect the current blue face
3. For every green face found in the above step find subvertices lying on the current blue face and the green face using the face intersection primitive

4.2.2.1 Step 1 - Extract projection of the blue vertices

As mentioned earlier we use the mesh B' in place of the blue mesh B for better robustness. From step one, we have the projections of all the blue vertices on the green mesh. In this step we extract the projection of all the three vertices of the blue face. This information is necessary for two reasons:

- the projections are subvertices with blue vertex as blue parent and its projection as green parent, and
- to find a list of green faces that could potentially host the subvertices in the interiors of the blue edges of the current face

For every blue face we keep a list of green faces that could potentially intersect with it. Green faces are added to this list as more subvertices on the blue face are found. To begin with, all the green faces that the vertices of the blue face projected onto in the first step are added to the list. These green face are the seed points which help in locating more green faces that could intersect with the current blue face. If a vertex of the blue face projected onto a vertex in the green mesh then all the faces incident on that green vertex are added to the list of potential candidates. On the other hand, if a blue vertex projected onto a blue edge, then both the faces (one face

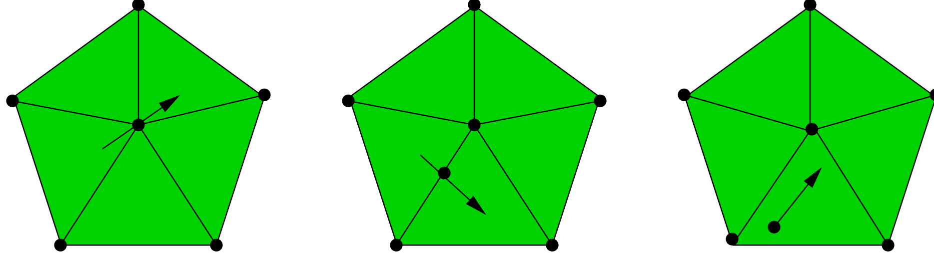


Figure 11: Adding potential green faces that could intersect with a blue face

if it is a boundary edge) incident on that edge are potential green faces. If a vertex falls inside a green face, then only that green face is added to the list because a vertex can just lie inside one green face at a time. Once we have our initial list, we apply the face-intersection primitive to the blue-green face pairs. As shown in the Figure 11, whenever a subvertex is found we add more green faces to the list depending on whether the green parent is a vertex (all faces incident on green vertex are added), an edge (the other green face incident on the edge is added) or a face (no green face is added). In this way we find all the subvertices lying on the blue face. This step does not use the kdtree data structure to find potential green faces that could intersect with the blue face. The information derived in step one gives us some seed faces and we build up on that information to find other green faces that could potentially intersect with the blue face.

4.2.2.2 Step two - Find subvertices using face intersection primitive

Procedure

This step uses the face intersection primitive to find subvertices. Using the relative position of projection of blue vertices found in step one, all the blue edge - green edge pairs for the current blue and green face pair are marked for which we already have an intersection. The edge intersection primitive is then applied to the rest of the blue edges and the green edges. Even though traversing the meshes face-by-face makes the algorithm easy to implement, it may lead to some duplicate computations. Some edge

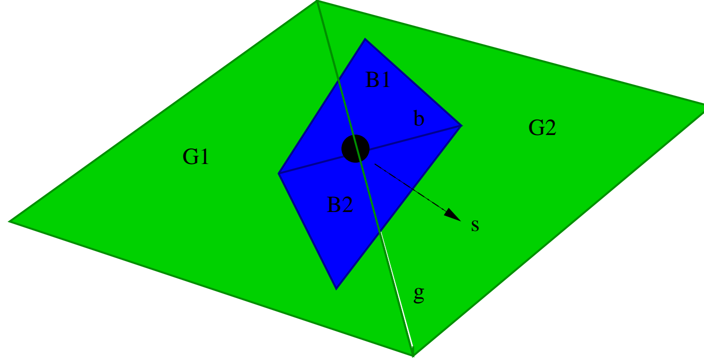


Figure 12: Duplication of blue-green edge pair intersection

pairs are considered more than once. For example, as shown in the Figure 12, the blue edge b will be tried for intersection with green edge g twice, once for green face $G1$ and then for green face $G2$. This would lead to duplicate copies of the subvertices lying on these edges. To tackle this problem, we never apply the edge intersection primitive to a pair of blue and green edge more than once. Whenever we get a pair to which we have already applied the primitive, we simply extract the old intersection

Implementation

Every vertex and edge of a face has local id with respect to a face. A face is represented as shown in Figure 13.

After the projection of blue vertices is extracted, their positions relative to the current green face is found. Also, the projection of green vertices is found in this step. For every blue and green face a 3×3 matrix is maintained, which keeps track of which combination of blue edge and green edge should be given to the edge intersection primitive. Entry at the position (i, j) of the matrix is non-zero if blue edge i and green edge j should be checked for intersection and zero otherwise, where i and j are the local ids of the edges in the face. This matrix is filled after getting information at two level:

(a) Face level: If a vertex in one mesh projects on a vertex in the other mesh, then the edges incident on them in both the meshes are not checked for intersection,

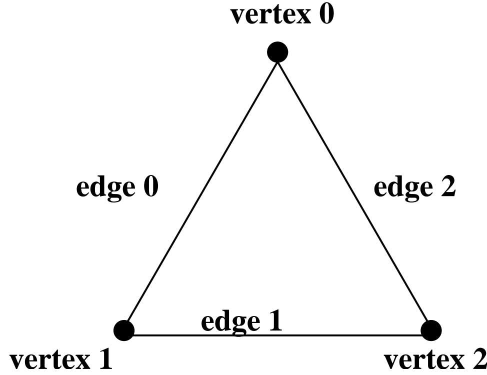


Figure 13: Representation of a face

because the two vertices are the points of intersection in their respective meshes. Also, if a vertex falls on an edge then all the edges incident on that vertex won't be checked for intersection with that edge for the same reason as above.

(b)Mesh Level: If a blue and green edge have already been checked for intersection (while computing subvertices for some other faces) then these edges are not checked again. Instead we extract the old subvertices. To make this possible for every edge we store the indices of the subvertex lying on it.

Also as subvertices are found for a particular green and blue face pair, we add more green faces to the list of the potential green faces that could intersect with the current blue face. If blue edge intersect with a green vertex then all the green face incident on that green vertex are added to the list of candidates. On the other hand, if the intersection of blue edge is a point in the interior of the green edge then we add the other face (if any) incident on that edge to the list. This follows from the host candidate lemma explained in [8]. It characterizes the topological relationship between the green parents of the subvertices in a blue edge b. The identification of potential green parents is very important for the efficiency and performance of the algorithm. On one hand we don't want to check for the intersection of all the blue faces and green faces in the two meshes but we also don't want to miss the green face that could have intersected with a blue face. If we fail to identify the green faces that

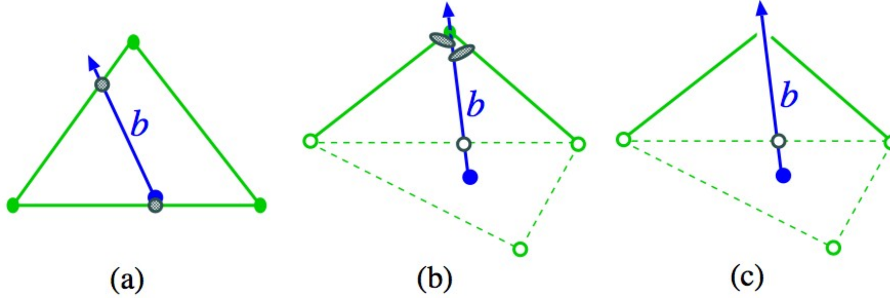


Figure 14: Different types of inconsistencies and their remedies

intersect with a blue face, then this would lead to holes in the common refinement of the two meshes.

4.2.3 Resolution of inconsistencies

The non-linear primitives involved in the algorithm can be solved only approximately. This leads to a inconsistencies in the subvertices found. Such inconsistencies must be identified and resolved properly for a robust, valid and accurate overlay of the two input meshes.

While locating subvertices on the edges of blue and green faces we compute the intersection of a blue edge with many green edges and vice versa. An edge might appear to intersect with more than one edge of the other mesh because of inexact computation of the primitives. We detect and resolve all such inconsistencies beacuse there is only one true intersection between two edges if they are not parallel.

As noted by Jiao anh Heath in [8], there are three different types of inconsistencies that can arise. The inconsistencies and their rememdies are shown in Figure 14. In the first two cases, numerical computations return two intersections between blue edge b and the green edges, with the diffrence that in Case (a) one of the intersection is close to (or at) the origin of b ($org\, b$), whereas in Case (b) both the intersections are far away from $org\, b$. Case (a) occurs when $org\, b$ is too close to one of the green edges. We peturb $org\, b$ onto its nearby green edge (by reassigning the green parent

of $org\, b$). Case (b) occurs when b is too close to the common vertex of the green edges that intersect b. (These green edges may not share a common vertes, but this would indiccate that the green mesh is not well shaped or the projection is far from orthogonal.) We resolve this inconsistency by peturbing the intersection to the green vertex. In Case (c), numerical errors cause b to fall into an artificial gap at a given green vertex between the green edges so that no edge intersection is reported. As for Case (b), this case happens only when b is too close to the green vertex, but Case (c) is more difficult because having no intersection is a valid solution. Case (c) differs from valid no-solution cases in that α is between 0 and 1 for some edge intersections rather than being always greater then 1. In Case (c), β is less than zero for the green edges to its left and greater than 1 for green edges to its right. We peturb the intersection to the green vertex if Case (c) is identified.

4.2.4 Making subfaces and Triangulation

Procedure

The subfacets are enumerated for the blue mesh and green mesh simultaneously. This step is broken down into two sub-steps as follows:

(a) Making polygonal subfaces

(b) Triangulating the polygons formed in the above step.

The second step above yields a list of subvertices and their blue and green parents in the two meshes. The next step is to make subfaces from these subvertices and subedges (intervals in the blue and green edges between the subvertices). The sorted lists of subvertices and their parent information is utilized to form the subfaces. The blue and green parents provide us with constraints we need to follow while making the subfaces. For a blue-green face pair we find out the common subvertices between them and try to form subface from those common subvertices. If two subvertices (from the list of common subvertices between a blue-green face pair) are on the same

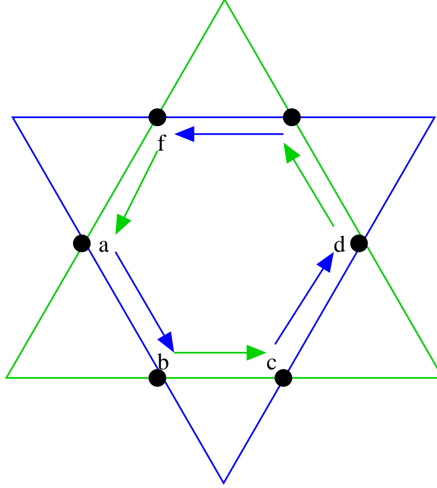


Figure 15: Subfaces using green and blue edge constraints. Blue arrows are blue constraints and green arrows are green constraints. Subfaces are made following the constraints of common subvertices between a blue and green face pair.

blue edge then they should be on the same edge in the subface too. This is a blue edge constraint. Similarly we have a green edge constraint. These constraints are applied to all the common subvertices between blue green face pair. For example, in Figure 15, a and b are on the same blue edge, which gives us a blue constraint. We start from subvertex a and go to b to satisfy the blue constraint. Then we need to find the next constraint for subvertex b. Subvertex b and c have a green edge constraint, so from subvertex b we move to subvertex c. In this way starting from the first common subvertex between a blue-green face pair we form a subface following the constraints by traversing the subvertices in counter-clockwise order.

Implementation

For this we first sort the subvertices on all the edges of a face for all the faces in both the meshes. Sorting is done based on the distance of a subvertex from the starting vertex of the face in counter-clockwise direction, with the subvertices lying farther away from the starting vertex found lower down in the sorted list than the subvertices lying near to the first vertex, in the sorted list as shown in the Figure 16.

Like all the previous steps we iterate through the blue mesh face by face and try to list the subfaces it hosts. Again, for this we need to find out the green faces that will host the subfaces with the blue face. These green faces can be easily found using the green parent information of the subvertices lying on the edges of the blue face. For every blue-green face pair we make a list of common subvertices between them. A vertex and its projection is also a common subvertex if the projection lies either inside the face on the other mesh or on one of it's edges or vertices. Once we have a list of common subvertices sorted in counter-clockwise order, we can proceed to make subvertices. If the previous step was correct, then the shared subvertices will have same cyclical order in both the lists (for blue face and green face). For the cases in which no vertex of one face projects inside the other face, the two lists have the same number of subvertices in same cyclical order. These sorted lists then provide the constraints with consecutive subvertices in the lists forming subedges. The subface is the patch bounded by the subedges formed by consecutive subvertices in the cyclical list. But for the cases when a vertex of one face projects inside the other face, the two lists of sorted subvertices are not of same length. After we have a subface we triangulate it to form the common-refinement mesh. We use the triangulation algorithm with ear removal [12] to triangulate the subfaces formed above.

4.2.5 Analysis of complexity

Let N_b and N_g be the number of faces in the blue and green mesh respectively. The first step traverses the nodal graph of B to locate subvertices in its incident edges. During the traversal we visit every blue face once and locate the subvertices lying on the edges of that blue face. This face traversal of the mesh guarantees that the total cost of the step is of the order of the number of faces in the blue mesh i.e. $O(N_b)$. The cost of building the kdtree is $O(N_g log N_g)$ which is spent just once at the beginning of the step. After that it is queried for every blue face. The cost of querying the kdtree

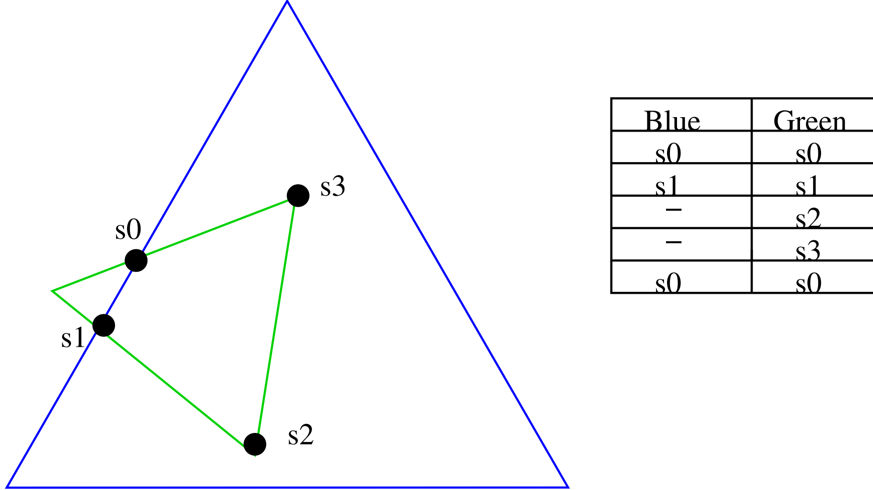


Figure 16: Sorting subvertices common to a blue-green face pair to form subfaces

once is $O(N_g^{1-\frac{1}{d}}+k)$, where k is the number of reported points and d is the dimension of the kdtree and is 3 for our algorithm. Parameter k represents the number of green faces that fit in the bounding box of the blue face and depends on the dimensions of the blue box and the relative resolution of the two meshes. So the cost of querying the kdtree for all the blue faces is $O(N_b(N_g^{\frac{2}{3}}+k))$. Similar to step, all the steps involve face traversal of the blue mesh. Therefore, all the other steps of the algorithm are $O(N_b)$.

4.3 Parallel Algorithm

The main motivation behind modifying the original algorithm was to make the process of computing common refinement easily parallelizable. With the advances in parallel computing, a lot of algorithms are being parallelized and they produce and use meshes that are partitioned across several processors. This generates the need to parallelize the computation of common refinement of two meshes. The largest conundrum that lies in the way of parallelization of any algorithm or process is the amount of interprocessor communication it entails. The way the meshes are partitioned generally does not guarantee that data required by a processor would always be present

on the same processor. Also in a lot of situations communication is inevitable, because other processors might need results or data produced by other processors. But interprocessor communication costs in parallel algorithms can greatly degrade the system performance and diminish the potential benefits of utilizing increased number of processors. As mentioned before, the meshes have been created and distributed across the processors independently. So there is no guarantee that the partitions from the two domains owned by a processor, overlap spatially i.e. if a processor owns a blue point p, then we cannot assume that the green element it would project onto would also be owned by it. So this processor should be aware of the green partitions owned by the other processors. This sort of global information is not available. For a processor to discover which processors own the parts of the green mesh that spatially overlap the blue partition owned by it, involves a lot of communication. Once a processor has received all those portions of the green mesh that could potentially overlap with its blue partition, then the computation of common refinement can be done serially using the serial algorithm defined in the previous section.

The following sections explains the parallel algorithm. The parallel common refinement computation algorithm consists of following five steps:

1. Discovering and receiving green mesh subsets
2. Computing subvertices locally
3. Resolving inconsistencies
4. Making subfaces
5. Redistributing subfaces

4.3.1 Step one - Discovering green mesh subsets

As stated previously, partitions of the two meshes owned by a processor do not represent the same geometric region of space as shown in Figure 17. To compute the

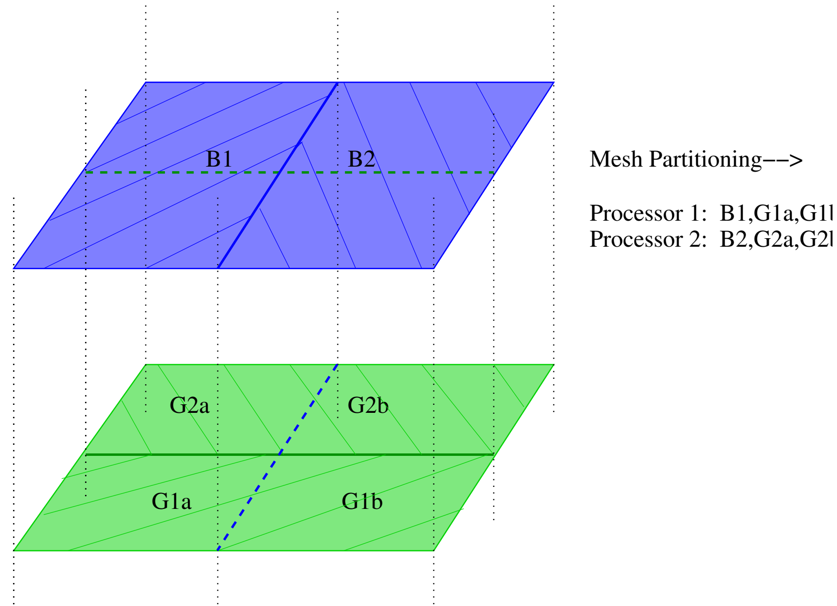


Figure 17: Mesh partitioning demonstrating that the blue and green partitions owned by a processor do not overlap

common refinement, a processor needs to be aware of the green facets that could intersect with the facets in its blue mesh. So we need a mechanism with which a processor would be able to request the other processors to send all such green facets to it.

A naive solution is for every processor to send its green partition to every other processor. Then all the processors can compute the subvertices locally. But such high interprocessor communication would curtail the speedup we are trying to achieve. We need a way to make sure that processor P_j only sends P_i those subsets f its green partition that are spatially overlap with the blue mesh of P_i. This is achieved by having every processor make a bounding box around all the connected components of its blue mesh, as shown in Figure 18 and send to the root process. The root process broadcasts this information to all the other processes. Now every process has the geometric extent of every other processor's blue partition. Using this information a processor P_j makes a list, for every other processor, of the green faces in it's partition that lie in the bounding boxes sent by P_i and hence, can intersect with it's blue faces.

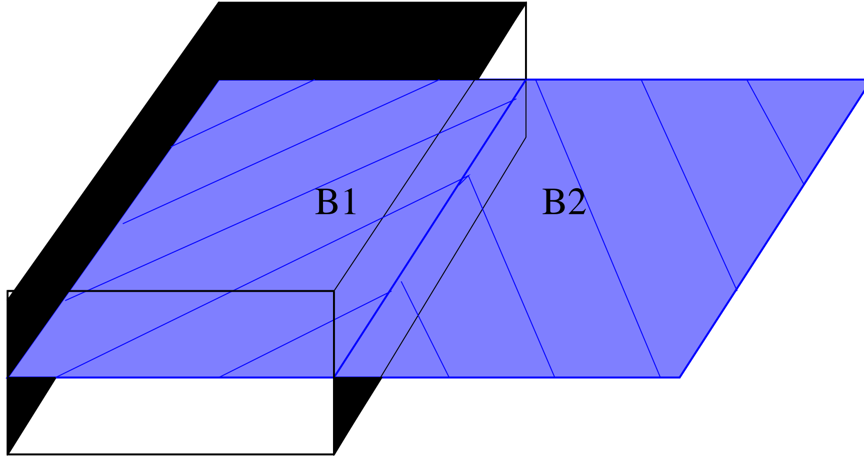


Figure 18: Every processor makes bounding boxes around its blue mesh partition to communicate its spatial extent to the other processors

This communication is done using non-blocking sends and receives. Once a processor receives the green faces that spatially overlap with its blue mesh partition, from the other processors, it can start computing subvertices independent of all the other processors.

In addition to these green faces a processor also sends some blue faces to the other processors. These are the blue faces lying on the boundaries of it blue mesh partition. It sends a blue boundary face to all those processors with which it shares the vertices of that blue face. This helps in the resolution of inconsistencies by a processor locally and obviates the need for communication between them while resolving inconsistencies on blue faces lying on the boundaries of the blue partitions.

4.3.2 Step 2 - Computing subvertices

In the first step all the information needed to compute the ssubvertices locally is collected by a processor. Once a processor has a partition of blue mesh and a list of all the green faces of the green mesh that could potentially intersect with it's blue mesh partition, locating subvertices can be done locally using the serial algorithm. This step again can be done in more than one ways, with each method having it's own advantages and disadvantages.

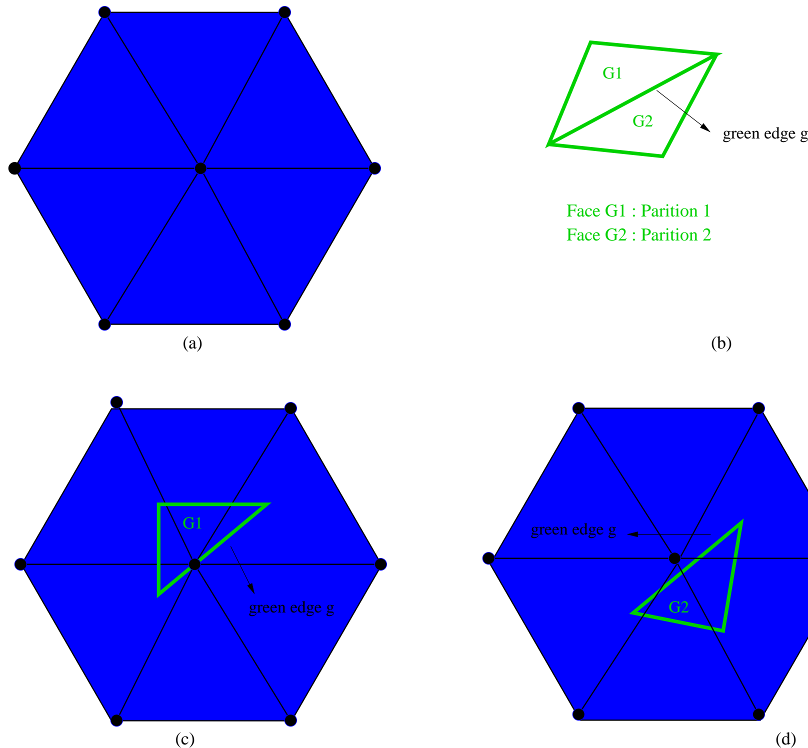


Figure 19: Inconsistencies that can arise along the green partition boundaries. Faces G1 and G2 have different subvertices lying on the same green edge g which they share. This arises due to numerical errors.

One of the strategies to compute the subvertices is to use the serial algorithm for every set of green faces received from the other processors, one at a time. The advantage of this method is that a processor can then overlap interprocessor communication with the computation step. But the biggest disadvantage is that it makes locating and resolving inconsistencies, especially along the green partition boundaries, extremely difficult, as shown in Figure 19. It necessitates the employment of a mechanism to find correspondence between the green elements received from different processors, which is a hard task. This defeats the goal of the new algorithm, namely, to simplify the computation of common refinement.

To overcome this disadvantage, we decide not to require such correspondence. It involves gathering all the green facets from different processors first, which are then numbered consistently using local numbering scheme of that processor. The

advantage gained with this method is that now no special treatment needs to be given to the subvertices lying on the facets on the boundaries of the green mesh partitions. The routines implemented for solving inconsistencies in the serial algorithm suffice for this parallel method too.

4.3.3 Step 3 - Resolving inconsistencies

Presence of numerical errors, which lead to inconsistencies in the computation of subvertices, is inescapable. They are resolved in the same way as in the serial algorithm. But the problem arises along the boundaries (both blue and green) because some coordination is needed between processors to resolve them. Inconsistencies along green mesh partition boundaries are tackled as explained in step one.

As shown in the Figure 20 green vertex gv1 falls on a blue boundary edge b for blue mesh partition B2 but does not fall on that edge for blue mesh partition B1. Both partitions need to communicate this to each other so that they can resolve this inconsistency. Resolution of this inconsistency might introduce changes in subvertices in the interior of the partitions which in turn could cause changes to the subvertices along the boundaries. Therefore, resolving inconsistencies in subvertices lying on the boundaries causes a lot of communication back and forth between processors till no changes are introduced and all of them settle down to a stable and consistent state.

But if both these partitions were aware of the blue face incident on blue edge b owned by the other processor then this can be resolved locally and without any further communication between the different processors. So, a processor collects all the blue faces owned by the other processors along its blue mesh boundaries.

As mentioned in step one a processor was sent all the blue faces lying on the boundaries it shares with other processor. Resolution of inconsistencies, along blue mesh boundaries, was done in the same way as the method adopted in the serial algorithm.

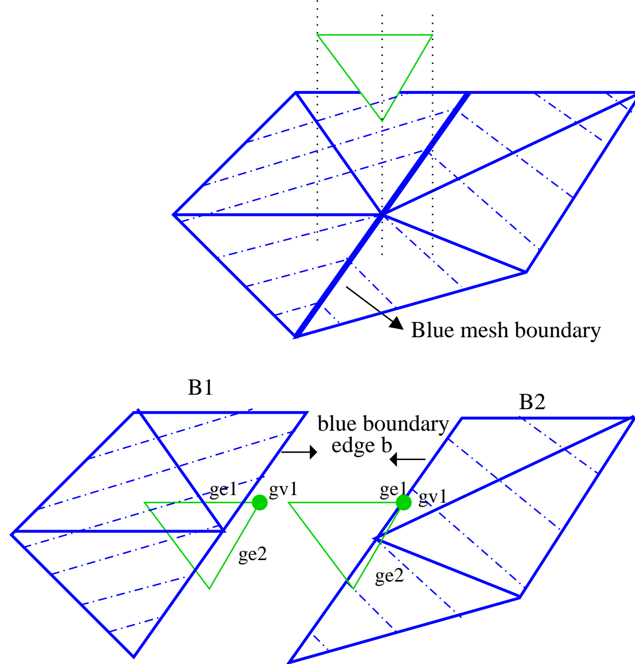


Figure 20: Inconsistencies along blue partition boundary

4.3.4 Step 4 - Making subfaces

After all the inconsistencies have been resolved, a processor can proceed to make subfaces from the subvertices, and triangulate them, just like in the serial algorithm. This step needs no communication between different processors.

4.3.5 Step 5 - Redistributing subfaces

After all the subvertices have been computed and the subfaces are created, a processor sends all the subvertices and subfaces for which the green parent face is the green face that it received from some other processor in step one to that processor. So in the end every processor has a list of thesubvertices and subfaces lying on it blue and green mesh partitions.

4.4 Comparison of algorithms

This section explains the basic differences between the original algorithm described in [8] and the new modified algorithm described in this thesis. As mentioned before

there were two main motives behind modifying the algorithm original algorithm to compute the common refinement:

1. Simplification of the implementation by introduction of simpler primitives and data structures

2. Parallelization of the algorithm

The main factor that makes the parallelization of the original algorithm challenging is the edge traversal of the meshes in the old algorithm. Traversal of the blue mesh edge-by-edge while locating subvertices, makes it inherently sequential. The old algorithm required the knowledge of green parent of one of the points in the blue mesh and from that point on, beginning with one of the edges incident on this point, subvertices were found while traversing the edges in breath-first order. Once the green parent of one of the points of an edge was known, the Host-candidate Lemma,[8], was used to find the potential green parents of the subvertices on that edge. This one green parent was the seed point to which the host candidate lemma could be applied to find green parents of the other points in the blue mesh. The new algorithm replaced edge traversal with face traversal and one seed point with a set of seed points (green parents of all the blue points) which simplified the computation of the subvertices.

Face traversal allowed replacing the lower level primitive by higher levels primitive, face-intersection. The main idea behind these higher level primitive like face intersection was to make parallelization simpler. The use of face intersection primitive simplified many of the data structures involved and also simplified the resolution of inconsistencies.

While using the face-intersection primitive, the inconsistencies could be resolved by using the knowledge of the location of the subvertices relative to their blue and green parents. Also it simplified the resolution along the boundaries of the partitions of blue and the green meshes used in parallel computing.

Introducing a higher-level primitives, as explained before, induced some redundant and duplicate calculations into the algorithm. To tackle this issue, for every subvertex lying on an edge a pointer to that subvertex was stored. Using this information all the redundant calculations were avoided.

As noted by the authors in [1], even though the original algorithm had rich potential for parallelism, there were three main challenges in parallelizing it. Those challenges are listed below:

1. The input meshes can be partitioned differently from each other, and hence one must redistribute at least one of the input meshes based on the partitioning of the other mesh: The new algorithm tries to beat these challenges as efficiently as possible. As illustrated in the section 3.4, the input green mesh is redistributed such that every processor receives the subsets of green mesh that overlap with it's blue mesh partition. To achieve this every processor sends the dimensions of bounding boxes of its blue mesh to all the other processors using which the processors decide which of the green faces owned by it are needed by that processor

2. To resolve inconsistencies caused by numerical errors near partition boundaries, some coordination is needed among processors: For inconsistencies along blue mesh boundaries a processor receives an extra layer of faces encompassing its boundaries from the other processors which makes the process of resolution of inconsistencies across blue mesh boundaries a local process and requires no communication between different processors. Collecting all the green faces from different processors, and then computing the subvertices, makes the resolution of inconsistencies along green mesh boundaries similar to the resolution serial algorithm.

3. Multiple numbering systems for vertices and facets are involved, which introduces additional complexities: To combat this all the green faces and blue faces received by a processor from the other processors are renumbered according to the local numbering system of a processor which greatly simplifies the entire process.

Another addition to the new algorithm is the support for partially overlapping meshes which are used often in real world problems. The old algorithm did not support partially overlapping meshes and assumed that the boundaries and features of the two meshes matched exactly. No such assumptions about the meshes are made in the new algorithm. The regions of the two meshes that have no correspondence in the other mesh are tagged by the algorithm.

CHAPTER V

RESULTS

Here, we illustrate the performance of the modified algorithm. The blue mesh used in the illustration contains 11,806 faces and 5,903 vertices. The green mesh has 45,460 faces and 22730 vertices. The meshes are shown in Figure 22. The serial code was ran on an intel 3.20 Ghz machine. It tool a total of 20 seconds to compute the overlay. The same serial code was also compiled and executed on SDSC's (San Deigo Super Computing Center) IA-64 Linux cluster with dual 1.5 GHz Intel processors. It took 24 seconds on these machines. The parallel code was ran on SDSC cluster too. The results are shown below.

No. of Processors	Time Taken (sec)	Speedup
2	15	1.6
4	5.92	4.054
8	3.24	7.407
16	1.61	14.90

As shown if Figure 21, the speedup achieved was almost linear.

For the parallel code, every processor owns a blue and a green subdomain. At the end of the computation, every processor writes two output files corresponding to the common refinement computed for each of the input meshes it owned. The output file provides following information

1. Local coordinates of the projection of all the vertices of one mesh, onto the other mesh except for the ones for which no projection was found.

2. For every subface it provides the global IDs of the realization of the subvertices, forming that subface, in the other mesh and their local coordinates in that mesh

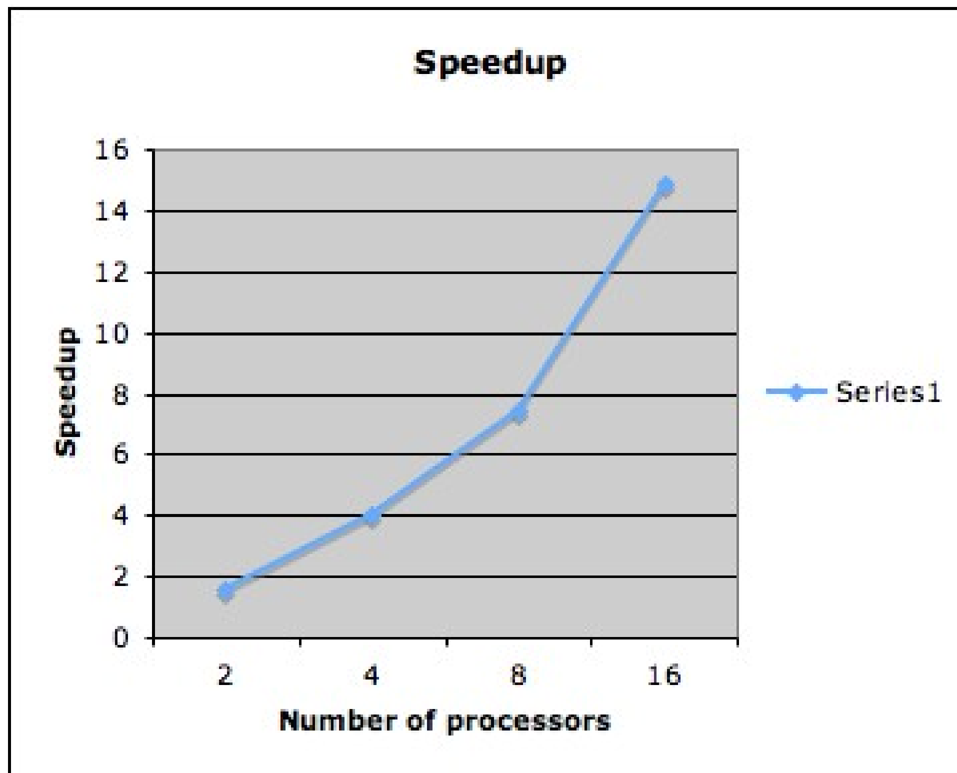


Figure 21: Speedup achieved for parallel code

3. For every subface, it also provides the corresponding ID of that subface on the other mesh.

Figures 23, 24, 25, 26, 27show the common refinement produced when the input meshes were partitioned into 4 subdomains and distributed across 4 different processors.

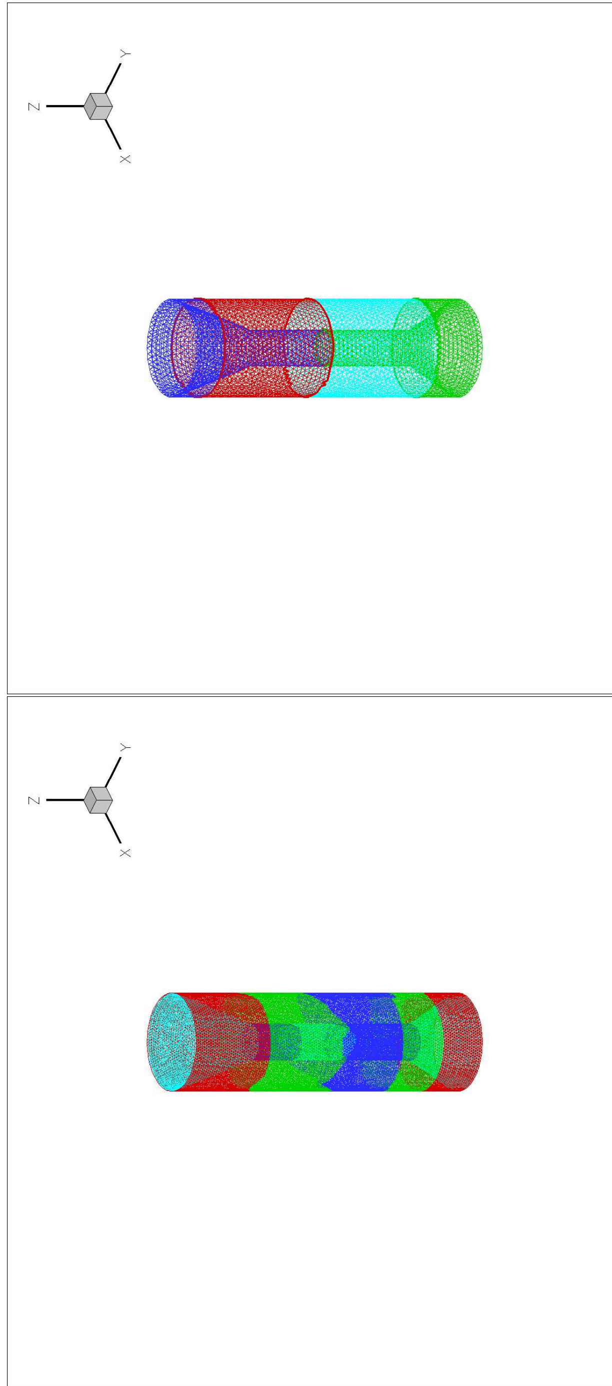
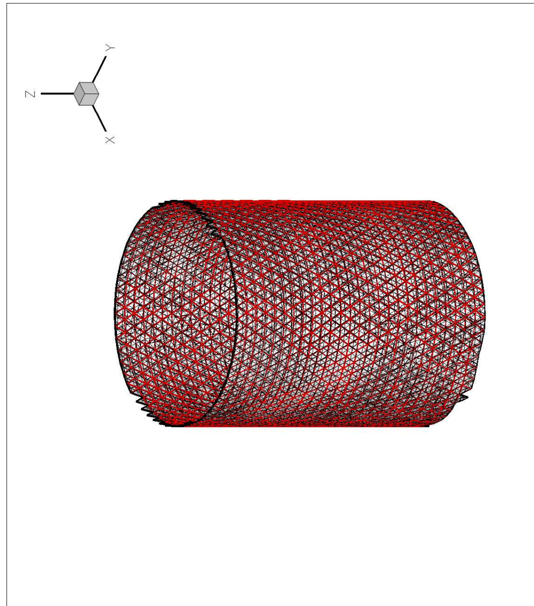


Figure 22: Blue mesh (above) and green mesh (below) used for illustration. The meshes have been divided into four subdomains with different colors representing different subdomains. Red, green, blue and cyan represent subdomains 1, 2 ,3 and 4 respectively.



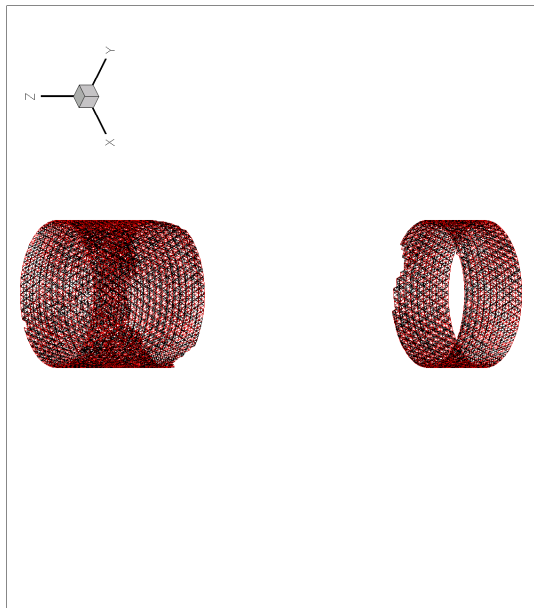
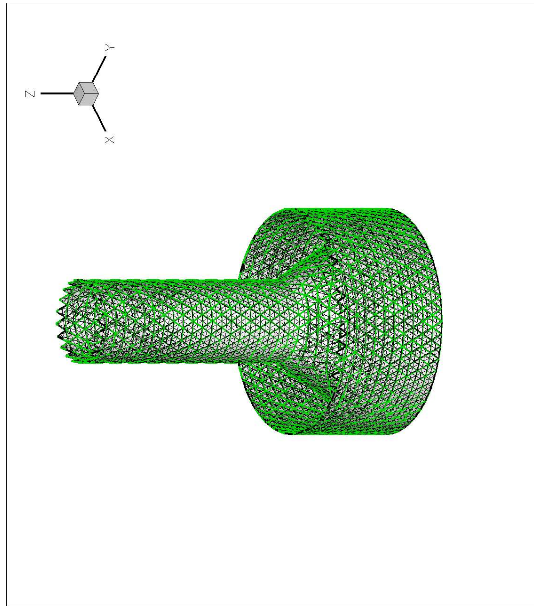


Figure 23: Input meshes (blue above, green below) and corresponding common refinement for processor 1



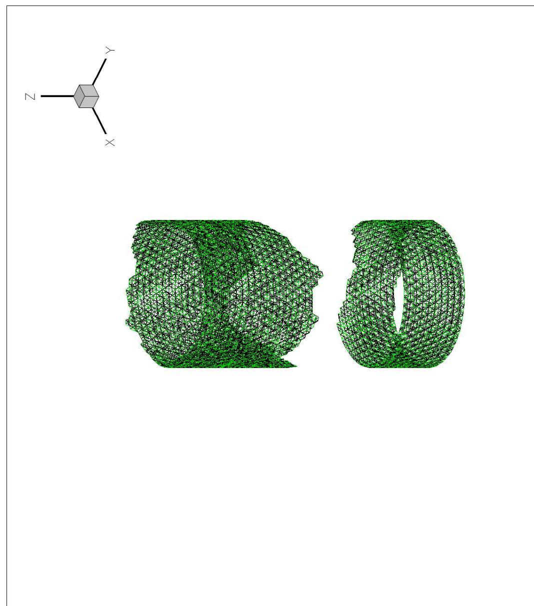
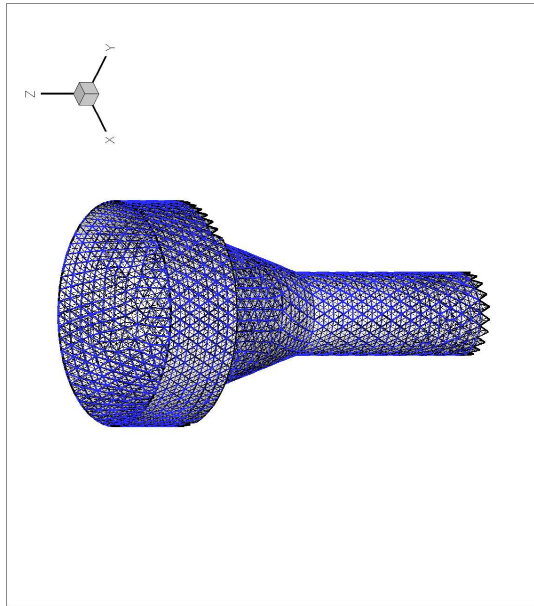


Figure 24: Input meshes (blue above, green below) and corresponding common refinement for processor 2



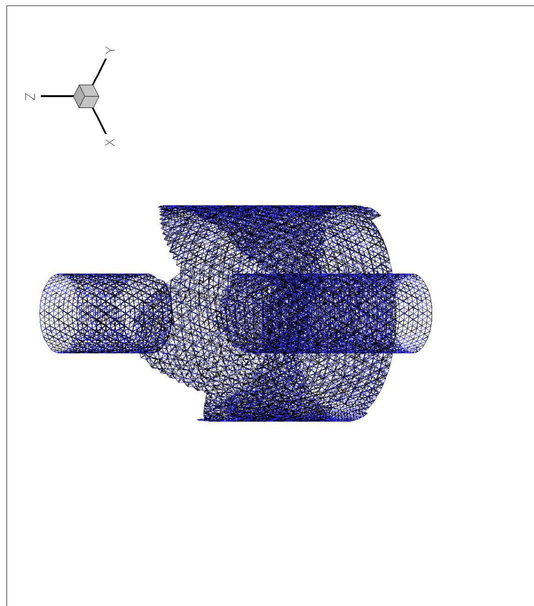
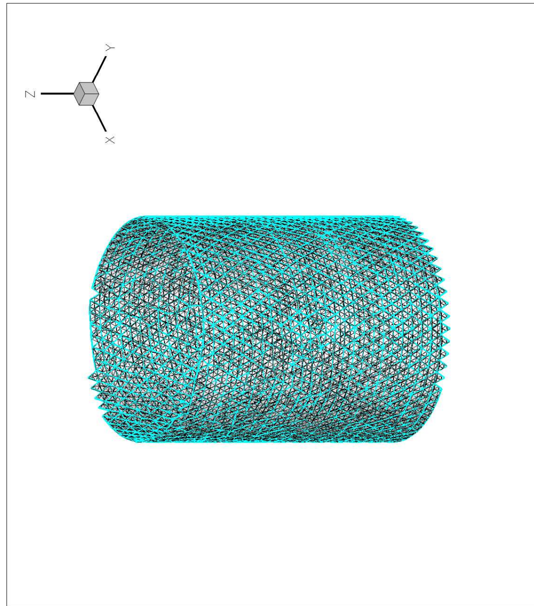


Figure 25: Input meshes (blue above, green below) and corresponding common refinement for processor 3



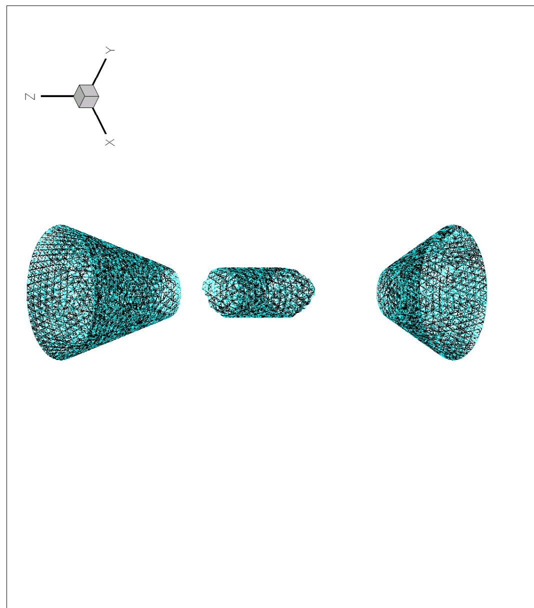


Figure 26: Input meshes (blue above, green below) and corresponding common refinement for processor 4

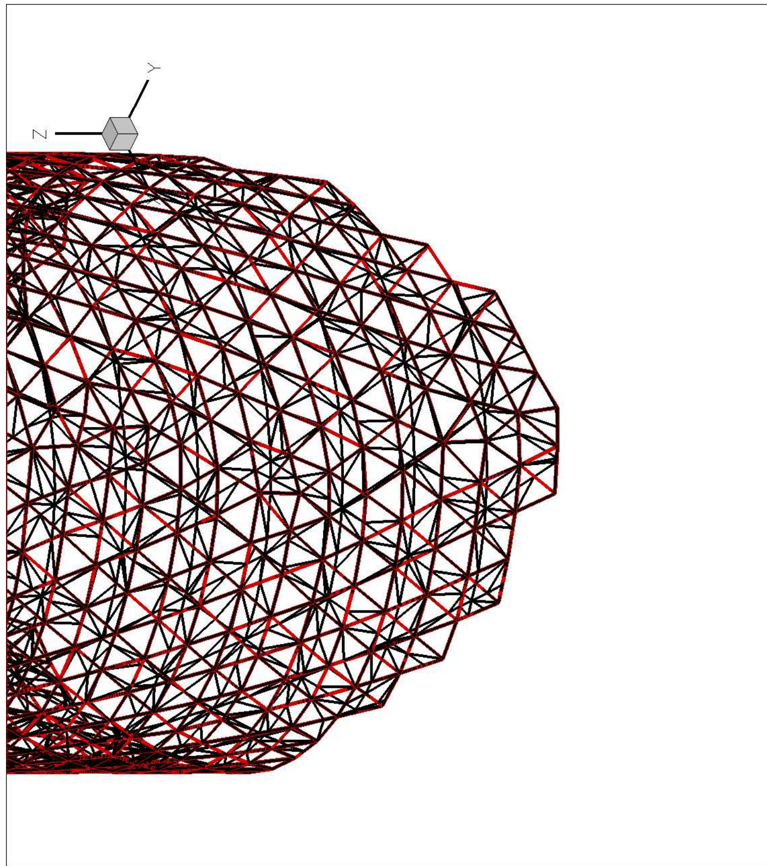


Figure 27: zoomed in view of common refinement

CHAPTER VI

APPLICATIONS AND CONCLUSION

6.1 Applications

This chapter lists some of the important applications of common refinement data structure.

6.1.1 Multi-physics Systems

Monolithic approach to solve multi physics systems involve using PDEs that govern the entire couples system are not deemed desirable because different physical components are governed by different mathematical and numerical properties and implementation codes. Some partitioned analysis techniques enjoy more popularity than the fully coupled monolithic approaches, as they allow the independent use of suitable discretization methods for physically and/or dynamically different partitions. Common refinement is a crucial data structure for transferring data between differing mesh representations of the surface models. Multi physics systems are omnipresent in real world applications. They are used in meteorology, climatic models, space weather, combustion, material science, hydrology, rocket engines, pharmaceutical, automative and many more. The biggest challenge in building these multi physics systems is the need for data transfer across the boundaries of the different components so that the system converges to an accurate solution. A very interesting multiphysics system application is MEMS. MEMS are mechanical devices coupled with electrical circuits having physical dimensions of micrometers. In these devices one physical phenomena converts to another e.g. electricity to motion and fluid pressure to electricity. MEMS devices involve strong coupling between structural parts, electrostatic field, fluid flow, thermal transfer and piezoelectric effects. Examples of MEMS devices include air bag

sensors, micro pumps, optics and hearing aids, ink jet printer heads, nerve simulations and many more. All these multi physics systems have one thing in common and that is the presence of interfaces between different components involved and the need to transfer data across these interface. Different components are discretized using different schemes that are best suited to them. So we need a of mapping among these different discretizations of an interface. Such a mapping can be developed using common refinement and used for data transfer.

6.1.2 Fluid-Solid Interaction

Fluid structure interaction (FSI) occurs when a fluid interacts with a solid structure, exerting pressure that may cause deformation in the structure and, thus, alter the domain of the fluid itself. e.g. the pressure exerted by the air on the wings of an airplane deforms the wings and this in turn changes the air pattern around the wing. Because of the basic differences between them the solid and the fluid domains, they are discretized using different schemes. So the common interface has two realizations, one at the fluid side and the other at the solid side. To solve the entire physical system we need to transfer data from the fluid side of the interface to the structure side of the interface. As mentioned before these meshes are non-matching. This is where the computation of common refinement helps. Analysis of fluid structure interaction is used in many engineering applications among them the following:

- Aerospace industry - To study foil flutter and constantly changing air pattern around the wings which in turn causes the air wings to flex and bend

- Welding analysis - involves several simultaneous effects: flow with heat transfer, phase change, thermal contraction during cooling which may result in residual stresses in the structures.

- Analysis of a breaking pipe - FSI analysis can be used to detect the leaking of pipes e.g. steam pipe of a boiling water nuclear reactor. The fluid domain is

composed of the region inside and outside the pipe and pipe comprises the solid domain.

- Other uses - Fluid-Structure Interaction analysis is useful for a wide range of applications such as fuel tank sloshing, drop testing of liquid filled containers, detonation wave effects on structures, pressure vessel analysis, fluid interaction with valves and elbows, explosively formed projectile (EFP) analysis, airbag deployment, parachute development, injection molding analysis, wind-force analysis on tall buildings, earthquake response of liquid storage tanks (nuclear waste disposal), oscillation of heat exchangers etc.

6.1.3 Graphical Applications

Many graphical techniques like morphing, shape blending ,transfer of texture or material properties and fitting template meshes to scan data require a one to one mapping between two or more models. Also it is required that shape and the features of the models is preserved by the mapping. Morphing is a special effect in graphics that involves smooth transition from one geometry into another. The most popular form of morphing involves creating an illusion of seamless transition from one person's face into another person's face to the viewer. The techniques for morphing find a correspondence between the two images while preserving the 3d shape. Care should be taken that the morph (correspondence between the two images) is not mathematically too different from the two given shapes. Morphing techniques have various applications ranging from special effects in television and movies to medical imaging and scientific visualization. In the typical morphing techniques the first image (source) is gradually contorted and it vanishes out, and the second image (target) starts to emerge. As the morphing proceeds, the first image (source) is gradually distorted and is faded out, while the second image (target) starts out and is faded in. Thus, the early images in the sequence are much like the first image. The middle image of the

sequence is the average of the first image distorted halfway towards the second one and the second image distorted halfway back towards the first one. The last images in the sequence are similar to the second one. A good morphing method should preserve the 3d shape i.e. it care should be taken that when similar 3d shapes are morphed that the resulting shape are not mathematically too different.

The problem is to determine the way in which the pixels in one image should be mapped to the pixels in the other image.This is where the common refinement algorithm can help. It can help to build the correspondence between the two images Another important graphics application is construction of 3D object models from the range scan of a physical object which again involves is determining the correspondences of points on one surface to points on the other. The common refinement algorithm can be used to find such correspondence because it works for non-matching meshes, too. Applications of construction of such 3D object models are: partial view completion, interpolation between scans, and recovery of articulated object models.

6.1.4 Biomedical Applications

Many important biomedical imaging techniques use separate meshes for measuring different quantities e.g. in EIT (Electrical impedance tomography) there is a need to measure different quantities like how the system reacts to external forcing and electrical potentials. These quantities are measured on separate meshes but to calculate the system in its entirety we need to transfer data among these meshes. Another example is measurement of data like electric field and composition of cardiovascular system which are measured at different instants of a cardiac cycle and are so represented on different meshes. So to develop any sort of correspondence between the data we need a correspondence between the meshes that are used to represent them. Another important area of application is analysis of blood flow in a diseased artery. The objective is to better understand the effects of a stenosis in the failure of a major

artery, which can lead to a heart attack. Here different meshes are used to model the blood flow and the diseases vessels and we need a way to find a mapping between the two meshes. Many bio-medical applications involve fluid flow and heat/mass transport in the body and in devices. Some examples include aerosol drug delivery, blood pumps, artificial heart valves, and diagnostic equipment. Such interactions between the devices and the human physiological flows are simulated for better design for the medical devices. Meshes are used in such simulations and again there is a need for finding a mapping between meshes from different domains. Separate meshes are used to study the fluid velocities, pressure, temperature of the fluid domain and stress, displacement,reaction forces on the structural domain (devices). Here too the common refinement can be used to transfer data at the interface of meshes from different domains.

6.2 Conclusion

In this thesis, we have proposed an algorithm which helps to build an overlay of two surface meshes that model the same surface. A lot of engineering applications comprise of myriad of complex interacting physical phenomenon that interact with each other at the interface. The interface is modelled by meshes that are geometrically and topologically discrepant, i.e. they are non-matching. The algorithm proposed,and implemented in the thesis is a modified version of the algorithm defined in [8]. The main motivation behind modifying the original algorithm was to exploit the locality of primitives involved in the computation of common refinement to make it easily parallelizable. The parallel algorithm retains the efficiency and robustness achieved for the serial algorithm.

The old algorithm didn't handle partially overlapping meshes which arise frequently in practice. Our new algorithm handles such partially overlapping meshes efficiently and tags the points in both the meshes that do not project to any face in

the other mesh.

Most of the meshes used in real problems model complex geometries. Such complex geometries induce sharp features such as ridges, corners and boundaries in the meshes. For most of the applications these features need to be matched and laid on top of each other. if mesh overlay is being used for data transfer in simulation of a multi physics system, then for accurate and reliable communication the features need to be matched. This sort of feature matching is also important in many graphical applications. For example. while morphing a horse to a pig, we would want to match legs to legs and arms to arms. Features matching is a two step process. First, we have to detect the features and then we need to match them. The algorithm in its current state does not employ any explicit feature detection and matching. There are some holes induced in the common refinement produced by the algorithm because of the lack of feature detection and matching. Jiao et al, in [9], have proposed techniques to detect and match geometric features. Such techniques can be easily embedded in our algorithm as preprocessing steps.

The algorithm uses green vertex normals while applying the primitive, point projection, to find projection of blue and green points on green and blue mesh respectively. The vertex normals are calculated as the weighted average of facet normals. If some form of smoothing is applied to these normal directions then the primitives would give better and more robust results along the features. The reliability of all the primitives used in the algorithm increases with the smoothness of the normals defined at the green vertices.

Also, the current algorithm uses kdtree data structure to find green faces that spatially overlap a blue face and hence can hold the projection of blue vertices of that blue face. These projections then are used as seed points in face intersection primitive to find more green faces that could intersect with the blue face. Even though this simplifies the entire process of computing subvertices, creating and querying kdtree

data structure is expensive. Due to time constraints, the current implementation still uses this data structure. As future work I plan to make it largely independent from it, which can be done by using locality search and would require few modifications.

REFERENCES

[1] Alexa, M., "Merging polyhedral shapes with scattered features," *The Visual Computer*, vol. 16, no. 1, pp. 26–37, 2000.

[2] Beckert, A., "Coupling fluid (cfd) and structural (fe) models using finite interpolation elements," *Aerospace Science and technology*, vol. 4, pp. 13–22(10), January 2000.

[3] Chazal, F., Lieutier, A., and Rossignac, J., "Orthomap: Homeomorphism-guaranteeing normal-projection map between surfaces," in *ACM Symposium on Solid and Physical Modeling (SPM)*, pp. 9–14, June 2005.

[4] Chazal, F., Lieutier, A., Rossignac, J., and Whited, B., "Ball-map: Homeomorphism between compatible surfaces," Tech. Rep. GIT-GVU-06-05, 2006.

[5] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O., *Computational Geometry – Algorithms and Applications*. Springer, second ed., February 2000.

[6] Guibas, L. and Seidel, R., "Computing convolutions by reciprocal search," in *SCG '86: Proceedings of the second annual symposium on Computational geometry*, (New York, NY, USA), pp. 90–99, ACM Press, 1986.

[7] Jiao, X. and Heath, M., "Common-refinement-based data transfer between nonmatching meshes in multiphysics simulations," *International Journal for Numerical Methods in Engineering*, vol. 61, pp. 2402–2427, December 2004.

[8] Jiao, X. and Heath, M., "Overlaying surface meshes, part i: algorithms," *International Journal on Computational Geometry and Applications*, vol. 14, pp. 379–402, December 2004.

[9] Jiao, X. and Heath, M., "Overlaying surface meshes, part ii: topology preservation and feature matching," *International Journal on Computational Geometry and Applications*, vol. 14, pp. 379–402, December 2004.

[10] Kraevoy, V. and Sheffer, A., "Cross-parameterization and compatible remeshing of 3d models," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 861–869, 2004.

[11] Maman, N. and Farhat, C., "Matching fluid and structure meshes for aeroelactic computations: A parallel approach," *Computers And Structures*, vol. 54, no. 4, pp. 779–785, 1995.

[12] O'Rourke, J., *Computational Geometry in C.* Cambridge University Press, second ed., 1998.

[13] Plimpton, S. J., Hendrickson, B., and Stewart, J. R., "A parallel rendezvous algorithm for interpolation between multiple grids," *J. Parallel Distrib. Comput.*, vol. 64, no. 2, pp. 266–276, 2004.

[14] Quaranta, G., Masarati, P., and Mantegazza, P., "A conservative meshfree approach for fluid structure interface problems," in *International Conference on Computational Methods for Coupled Problems in Science and Engineering*, (Santorini, Greece), May 24-27 2005.

[15] Schreiner, J., Asirvatham, A., Praun, E., and Hoppe, H., "Inter-surface mapping," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, (New York, NY, USA), pp. 870–877, ACM Press, 2004.

[16] Turk, G. and Levoy, M., "Zippered polygon meshes from range images," in *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 311–318, ACM Press, 1994.

[17] van Brummelen, E., "Mesh association by projection along smoothed-normal-vector fields: Association of closed manifolds," Tech. Rep. 1574-6992, Delft Aerospace Computational Science(DACS), Kluyverweg 1, 2629HS Delft, The Netherlands, june 2006.