MEMORY-EFFICIENT CONSTRAINED DELAUNAY TETRAHEDRALIZATION OF LARGE THREE-DIMENSIONAL TRIANGULAR MESHES

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Ziya Erkoç

July 2022

Memory-efficient Constrained Delaunay Tetrahedralization of Large Three-dimensional Triangular Meshes
By Ziya Erkoç
July 2022

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Uğur Güdükbay(Advisor)

Özgür Ulusoy

Ahmet Oğuz Akyüz

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

MEMORY-EFFICIENT CONSTRAINED DELAUNAY TETRAHEDRALIZATION OF LARGE THREE-DIMENSIONAL TRIANGULAR MESHES

Ziya Erkoç
M.S. in Computer Engineering
Advisor: Uğur Güdükbay
July 2022

We propose a divide-and-conquer algorithm that can solve the Constrained Delaunay Tetrahedralization (CDT) problem. It consists of three stages: *Input Partitioning*, *Surface Closure*, and *Merge*. We first partition the input into several pieces to reduce the problem size. We apply 2D Triangulation to close the open boundaries to make new pieces watertight. Each piece is then sent to TetGen [Hang Si, "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator", ACM Transactions on Mathematical Software, Vol. 41, No. 2, Article No. 11, 36 pages, January 2015] for processing. We finally merge each tetrahedral mesh to calculate the final solution. In addition, we apply post-processing to remove vertices we introduced during the input partitioning stage to preserve the input triangles. An alternative approach that does not insert new vertices and eliminates the need for post-processing is also possible but not robust. The benefit of our method is that it can reduce memory usage or increase the speed of the process. It can even tetrahedralize meshes that TetGen cannot do due to the memory's insufficiency. We also observe that this method can increase the overall tetrahedral mesh quality.

Keywords: Constrained Delaunay Triangulation (CDT), tetrahedralization, parallelization, three-dimensional triangular mesh, divide-and-conquer, Principal Component Analysis (PCA).

ÖZET

BÜYÜK ÜÇ BOYUTLU ÜÇGENSEL MODELLERİN BELLEK VERİMLİ, KISITLI DELAUNAY DÖRTYÜZLEMESİ

Ziya Erkoç
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Uğur Güdükbay
Temmuz 2022

Kısıtlı Delaunay Üçgenleme (KDÜ) problemini çözebilen bir böl-ve-yönet algoritması öneriyoruz. Algoritmamız üç aşamadan oluşmaktadır: *Girdi Bölme*, *Yüzey Kapatma*, ve *Birleştirme*. Problemin boyutunu küçültmek için önce girdiyi birkaç parçaya bölüyoruz. Yeni parçaları su geçirmez hale getirmek adına açık yüzeyleri kapatmak için 2D Üçgenleme uyguluyoruz. Her parça daha sonra işlenmek üzere TetGen [Hang Si, "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator", ACM Transactions on Mathematical Software, Cilt 41, Sayı 2, Makale No. 11, 36 sayfa, Ocak 2015] programına gönderiyoruz. Sonunda, nihai çözümü hesaplamak için her bir dörtyüzlü ağı birleştiriyoruz. Ek olarak, girdi üçgenlerini korumak için girdi bölme aşamasında eklediğimiz köşeleri kaldırma işlemi uyguluyoruz. Yeni köşe eklemeyen ve de köşeleri geri silme işlemini ortadan kaldıran alternatif bir yaklaşım da mümkündür; ancak, bu yaklaşım her zaman doğru bir şekilde çalışmamaktadır. Yöntemimizin yararı, bellek kullanımını azaltabilmesi ya da işlemin hızının artırabilmesidir. Yöntemimiz TetGen'in bellek yetersizliğinden dolayı yapamadığı girdileri başarı ile işleyebilmektedir. Ayrıca, bu yöntemin dörtyüzlü ağ kalitesini artırabildiğini de gözlemledik.

Anahtar sözcükler: Kısıtlı Delaunay Üçgenleme (KDÜ), dörtyüzlüleştirme, paralelleştirme, üç boyutlu üçgensel model, böl-ve-yönet, Temel Bileşenler Analizi.

Acknowledgement

I would like to thank my advisor Prof. Dr. Uğur Güdükbay for introducing me to the Computer Graphics field and his invaluable guidance throughout my research. I sincerely appreciate Prof. Dr. Özgür Ulusoy and Prof. Dr. Ahmet Oğuz Akyüz for kindly accepting to be my thesis jury.

I sincerely acknowledge Dr. Hang Si of Weierstrass Institute for Applied Analysis and Stochastics (WIAS), Berlin, Germany, for collaborating with us in this research, sharing the source code of his famous tetrahedralization software TetGen, and providing helpful feedback for my research.

I am grateful to Aytek Aman, Serkan Demirci, and Sinan Sonlu for fruitful discussions and comments and for providing feedback on my research.

I would like to thank all my friends for being helpful and supportive during my thesis study. Finally, my greatest thanks to my family. I am forever grateful.

The Bunny and Armadillo are obtained from The Stanford 3D Scanning Repository. The Neptune model is courtesy of Laurent Saboret by the AIM@SHAPE-VISIONAIR Shape Repository. The Spot, Blub, Bob, Pittsburgh Bridge, and Nefertiti models are courtesy of Keenan Crane's 3D Model Repository.

This research is supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. 117E881.

Contents

1 Introduction **1**

2 Related Works **4**

2.1 Constrained Delaunay Triangulation 4

2.2 Parallel Delaunay Triangulation 5

2.3 Input Partitioning 8

3 Proposed Approach **9**

3.1 Overview 9

3.2 Input Partitioning 10

3.3 Surface Closure 12

3.4 Merge 18

4 Modes of the Algorithm **23**

4.1 Memory Requirement Reduction 23

4.2 Parallel Processing 24

5 Experimental Results **25**

5.1 Mesh Quality 25

5.2 Parallel Processing 26

5.3 Memory Requirements 29

6 Alternative Approach **31**

6.1 Overview 31

6.2 Partitioning Stage 32

6.3 Surface Closure Stage 33

6.4 Repairing Stage 34

6.5 Limitations of the Alternative Approach 40

7 Conclusion **41**

Bibliography **43**

List of Figures

3.1 The illustration of intersection and point insertion. 12

3.2 The illustration of the refinement stage on the Armadillo model. . 15

3.3 The illustration of the parts after undergoing the surface closure stage. 16

3.4 The illustration of handling inputs with various topological structures during input partitioning (first part). 17

3.5 The illustration of handling inputs with various topological structures during input partitioning (second part). 18

3.6 The 2D view of example boundary polygons generated after surface partitioning. 19

3.7 The input partitioning and surface closure stages are illustrated on Bunny (top row) and Armadillo objects (bottom row). 19

3.8 Example tetrahedral meshes generated with our implementation. . 21

3.9 Example tetrahedral mesh outputs illustrating the result of the postprocessing step. Postprocessing is enabled in the middle image but not in the left image. The right image is the result of TetGen. All extra vertices on the boundary are removed, and the constrained faces are faithful to the input mesh. Our result with postprocessing enabled is identical to TetGen's output, which is the right image. 22

5.1 Execution time dissection for Armadillo, Bunny, and Nefertiti objects. 27

6.1 Illustration of partitioning algorithm applied on Torus Knot object. 32

6.2 Illustration of the left mesh with its boundary polygons marked in orange. 35

6.3 Illustration of the surface closure algorithm. 36

6.4 Illustration of the left mesh with its boundaries closed. 37

6.5 Illustration of the TetGen's diagnosis speed-up process. 38

6.6 Handling the degenerate case. 39

6.7 Handling edge-face intersections. 40

List of Tables

5.1 Vertex and face counts of the objects used in the experiments. . . 25

5.2 Experiments on tetrahedral mesh quality. 26

5.3 Experiments on the effects of post-processing (i.e., vertex removal) step on tetrahedral mesh quality. 28

5.4 The effect of density control parameter on the execution time. . . 29

5.5 Memory usage (in MB) and processing times (in seconds) for various models with post-processing enabled. 29

5.6 The effect of density control parameter on the memory usage. . . 30

Chapter 1

Introduction

Tetrahedral meshes are widely used in many areas, including bioengineering [1], biomechanics [2, 3], Computational Fluid Dynamics [4], computer graphics, and animation, especially the simulation of deformable bodies, including fracture and incision simulations [5, 6, 7], mechanical simulations such as turbomachinery flow [8], medical applications, such as medical device design [9, 10], medical image analysis [11], and soft tissue simulations [12]. Tetrahedral meshes are mainly used for the discretization of continuous materials and objects for Finite Element (FEM) Simulations [13] because they fit well for complex geometry [14]. Tetrahedralization algorithms in the literature solve various problems associated with tetrahedral meshes, such as preventing self-intersections and element inversion [15], reducing sliver tetrahedra for mesh optimization [16, 17], and approximations using point insertion to overcome the problems preventing constrainedness [18].

In the Computational Geometry domain, two-dimensional (2D) triangulation means forming triangles out of a co-planar vertex set. On the other hand, three-dimensional (3D) triangulation algorithms take inputs in 3D space and create a tetrahedral mesh. Delaunay triangulation algorithms form triangles that are close to an equilateral triangle. Such algorithms ensure that each triangle is approximate to an equilateral triangle and complies with the Delaunay criterion.

Our focus is 3D Constrained Delaunay Triangulation (CDT); i.e., Constrained Delaunay Tetrahedralization. These algorithms input a 3D point cloud and a triangulated surface mesh around the point cloud, and they generate tetrahedral meshes for the 3D point cloud whose boundary is the given 3D triangular mesh.

We observe that recent 3D CDT algorithms may suffer from that sufficiency of memory or may run slow [19, 20, 21]. When the memory is limited, these algorithms may fail to tetrahedralize an object because of excessive memory usage. We want to develop a tetrahedralization framework that would allow either small memory usage or fast execution. In this way, we could tetrahedralize substantial objects that other algorithms cannot do. In addition, we aim to speed up the process using parallelization structures. To this end, we introduce a divide-and-conquer algorithm.

Our divide-and-conquer algorithm is composed of three stages: *Input Partitioning*, *Surface Closure*, and *Merge*. To divide the input into smaller subproblems, we partition the input into pieces using parallel planes. We apply Principal Component Analysis (PCA) to find the most dominant axis to divide the input into as many pieces as possible. To create a watertight mesh, which TetGen requires, we triangulate open boundaries during Surface Closure. With the watertight meshes ready, we can execute TetGen for each piece. After the executions are complete, we merge the tetrahedral meshes and remove the extra vertices introduced during the input partitioning stage.

Our method has *memory optimization* and *parallel processing* modes. In memory optimization mode, we sequentially process the inputs in the former to reduce the memory footprint. In addition, we save intermediate files to disk to reduce memory consumption. In the parallel processing mode, we apply multi-threading to process pieces in a parallel fashion. Our results suggest that we can increase the speed or reduce the memory usage but not both simultaneously. We can increase the tetrahedral mesh quality by refining the triangles generated during the surface closure stage.

We also experimented with an alternative approach that does not introduce

new vertices at the beginning. However, this approach is not robust because of the following reasons. Projecting a 3D polyline onto the 2D plane required for closing the open boundary of the parts obtained by partitioning is problematic because this projection is not bijective and cannot ensure that the triangles generated during this process will not self-intersect in 3D. Additionally, self-intersection tests to detect such self-intersections are not reliable.

The contributions of this thesis are as follows.

- a divide-and-conquer algorithm to reduce memory usage and speed up the process,
- a parallelization scheme to run tetrahedralization in a multi-threaded fashion, and
- a straightforward PCA-based partitioning to allow for balanced partitioning of the input mesh in terms of the number of vertices.

The thesis is organized as follows. Chapter 2 discusses related work on triangulation and tetrahedralization algorithms. Chapter 3 describes our approach by explaining how we divide the input mesh, triangulate the open boundary of each part by a surface closure algorithm, and merge the sub-problems. Chapter 4 talks about the two modes of our algorithm in detail. Chapter 5 presents the experimental evaluation of the proposed approach in terms of execution time, memory usage, and mesh quality. We introduce an alternative approach that we experienced in Chapter 6. Finally, Chapter 7 gives conclusions and future research directions.

Chapter 2

Related Works

2.1 Constrained Delaunay Triangulation

Si put forth a CDT algorithm called *TetGen*, which generates high-quality tetrahedra, and his algorithm works quite fast [19]. Because the algorithm does not apply problem partitioning, it may not scale to large objects due to large memory requirements. Our algorithm is a memory-efficient, divide-and-conquer extension of *TetGen*.

Hu et al. [20] developed a robust CDT mesh generator called *TetWild*. *TetWild* is quite powerful because it can tetrahedralize a wide range of objects. It does not make input assumptions and can even process non-manifold objects with self-intersections. Because our algorithm uses *TetGen* at the base, the input models that we can handle are watertight non-self-intersecting meshes. *TetWild* algorithm is an approximate constrained algorithm; it might not preserve the input perfectly. Still, it controls the input preservation level with a parameter.

Chew proposes a two-dimensional sequential, high-quality constrained tetrahedralization algorithm [22]. The algorithm ensures that angles of the triangles are between 30 and 120 degree and edges are between h and $2h$ where h is a

user-defined value. These properties are guaranteed to make sure triangulation contains near-equilateral triangles. The algorithm constantly computes triangulation and finds a Delaunay circle, a circumcircle of a Delaunay triangle, whose radius is greater than h. It then inserts the circle's center as the new point and recomputes the triangulation. *TetGen* also uses a similar approach by adding a circumcenter of the poor-quality tetrahedra to increase the mesh quality [19].

2.2 Parallel Delaunay Triangulation

Although there are various studies on parallel two-dimensional Constrained Delaunay Triangulation (CDT) or parallel three-dimensional (3D) Delaunay triangulation (DT), we did not come across any parallel 3D CDT algorithms.

Chernikov and Chrisochoides's parallel 2D CDT algorithm uses a domain decomposition method called Medial Axis Domain Decomposition (MADD). After the decomposition, each subdomain is tetrahedralized in parallel independently. The number of sub-domains created is much higher than the number of processors, and they use the Load Balancing library to assign sub-domains to processors in the most flexible way possible. In addition, they solve a Graph Partition problem to distribute sub-domains to processors so that each processor has a nearly equal amount of work. They use the message-passing model as their parallelization scheme instead of a shared-memory structure, and their implementation is based on Message-Passing-Interface (MPI) library [23]. The domain decomposition algorithm presented here may work well in the 2D case, but in 3D, the existence of faces would make the problem more complicated. Hence, we used mesh cutting to decompose the domain [24, 25].

Coll and Guerrieri propose a 2D CDT algorithm that is parallelized using GPUs. Their algorithm comprises Location, Insertion, Marking, and Flipping stages. During the Walking phase, they identify triangles containing an uninserted point. They insert points in the Insertion stage. During the Marking

stage, the segments are marked either as valid or to-be-flipped (i.e., to eradicate non-Delaunayness or intersection). In the Flipping stage, they flip the edges marked so. The algorithm is an iterative one that continues to run these four stages as long as some PSLG elements (i.e., edges and points) are missing in the triangulation. These four stages are run one after another; they applied parallelization within each stage. In their implementation, the threads coordinate to avoid race conditions. For instance, when a thread is about to do a point insertion or edge-flip to a triangle, it informs the neighbor threads of that operation and who will be their new neighbor. It would be possible to extend this algorithm to be a 3D algorithm. We adopted a more straightforward approach by creating independent parts and processing them individually. That way, we ensure no race condition, synchronization issue, or thread communication, helping to reduce parallelization overhead [26].

Blandford et al. [27] propose a parallel tetrahedralization algorithm that can be extended to an out-of-core algorithm. However, it is not a constrained tetrahedralization algorithm. They suggest that developing an out-of-core algorithm would allow large meshes to be tetrahedralized. Their parallel algorithm is based on the sequential incremental insertion algorithm. They use multi-threading and lock mechanisms to insert multiple vertices into the tetrahedral mesh simultaneously. Our algorithm differs from theirs because we use the divide-and-conquer paradigm to tetrahedralize. Their algorithm does not divide the input as in our case but works on a single mesh with multiple threads. Chernikov and Chrisochoides also generated quality tetrahedral meshes using the circumradius-to-shortest-edge ratio as the quality measure [28]. Their algorithm leverages multi-core processors through parallelization. Specifically, they focused on parallelizing the Delaunay refinement step to speed up the overall process.

Cignoni et al. [29] put forward a divide-and-conquer Delaunay triangulation algorithm, *DeWall*, that can triangulate meshes of any dimension. Although it is not implemented as a parallel algorithm, it is amenable to a parallel implementation. However, it is not a constrained tetrahedralization algorithm.

Our divide-and-conquer algorithm differs from *DeWall* in the non-recursive

part. *DeWall* applies a merge step before the recursive step. This early-merge step uses a dividing plane and selects the vertices at either side of this plane to create an initial tetrahedralization. It chooses these vertices so that the generated tetrahedra have the smallest circumsphere radius to satisfy the Delaunay criterion. At this early merge step, the generated tetrahedra intersect the dividing plane. Then, it applies the same procedure recursively for the parts on either side of the dividing plane. We do not allocate buffer regions; we divide the mesh into parts and process them. Specifically, *DeWall* tetrahedralizes three pieces at each recursive step: the *DeWall* region around the dividing plane, left and right parts. We apply tetrahedralization to each part and do not spare a volume in the middle. The disadvantage of not reserving a middle region is that we cannot guarantee the Delaunay property for the tetrahedra around the cutting plane. Our rationale for not adopting this approach is not to slow down the process. Further, they could do this wall generation as part of a non-constrained triangulation algorithm but applying the same for a CDT algorithm might cause difficulties because a CDT algorithm must preserve the surface faces.

Chen et al. [30] proposed a parallel non-constrained near Delaunay triangulation algorithm. They divided the input into m blocks containing a nearly equal number of vertices. They triangulate each block using a divide-and-conquer algorithm. They call the area between these blocks as *interface*. These interfaces are built incrementally, applying a similar algorithm as used in *DeWall* to create the middle region. The middle-region creation is similar to *DeWall* and different from our algorithm because we do not spare a middle region but divide the input mesh into several pieces directly.

Marot et al. [31] came up with a parallel 3D Delaunay Triangulation algorithm. Their Moore curve-based input partitioning allows different threads to work on different sets of vertices. They allocated a buffer zone between partitions to fix potential conflicts raised by multiple threads.

Hu et al. [21] later developed a faster version of *TetWild*, called *fTetWild*. It is as robust as the *TetWild* but at the same time significantly faster. They used parallelization structures to accelerate their algorithm.

2.3 Input Partitioning

We shall discuss input partitioning techniques for triangulation algorithms to divide the problem into smaller subproblems. Joshi and Ourselind propose a constrained tetrahedralization algorithm that uses 3D convex decomposition and BSP trees [32]. They decompose the whole object into convex sub-polyhedra, tetrahedralize each piece and merge the meshes at the end. They accelerate the merge process using BSP trees. During the construction of the BSP tree, their algorithm introduces new vertices on the boundary. The largest model they used to conduct their experiments contains 26 vertices. We did not consider such an approach because we cannot control the number of convex sub-polyhedra generated and might subdivide the problem redundantly. One problem with redundantly subdividing is that the overhead of merging at each step might significantly slow down the process. To this end, we divide the object into a user-defined number of pieces.

Smolik and Skala suggest a 3D triangulation algorithm that divides the input into a 3D Grid [33]. They extended their algorithm to be out-of-core so that the memory usage is reduced and large scenes can be tetrahedralized. However, their algorithm is not constrained. They accept a vertex cloud as input and embed each vertex to a cell in the 3D grid. After that, they triangulate each cell and merge them at the end cleverly to complete the algorithm. We could not apply that approach as we cannot divide the input surface mesh into a regular 3D grid. Triangles might be present in multiple grid cells, which makes it difficult tetrahedralize each cell. Therefore, we separate the object into several pieces instead of using a grid structure.

Erkoc et al. [34] developed a divide-and-conquer constrained Delaunay tetrahedralization algorithm. Like our algorithm, they recursively divide the initial model into two pieces at each step and call the *TetGen* as the base case. Unlike our algorithm, they do not introduce any parallelization structure. In addition, they do not propose any plane selection algorithm and introduce costly repairing and merging steps.

Chapter 3

Proposed Approach

3.1 Overview

We propose a divide-and-conquer CDT algorithm that is composed of three stages: *Input Partitioning*, *Surface Closure*, and *Merge* (cf. Algorithm 1). In the first stage, we divide the input into smaller pieces. Specifically, we select parallel planes perpendicular to the most variant axis corresponding to the first principal component. We intersect those planes with the input mesh and insert new points at the intersection points so the mesh can be easily divided. Secondly, we apply 2D triangulation to close open surfaces to make the sub-pieces a watertight mesh that TetGen requires. We process each sub-piece using TetGen and finally merge the tetrahedral meshes. We also apply postprocessing to delete vertices introduced earlier during this stage.

Algorithm 1 Proposed Algorithm

```
 1: procedure TETRAHEDRALIZE(mesh, k, density_factor)
 2:     meshes, planes = INPUT_PARTITIONING(mesh, k)
 3:     for i = 0; i < meshes.size(); i++ do
 4:         CLOSE_SURFACE(meshes[i], meshes[i + 1],
 5:             planes[i], density_factor)
 6:     end for
 7:     tetmeshes = []
 8:     for i = 0; i < meshes.size(); i++ do
 9:         tetmeshes[i] = TetGen(meshes[i])
10:     end for
11:     tetmesh = merge(tetmeshes)
12:     return tetmesh
13: end procedure
```

3.2 Input Partitioning

The proposed algorithm begins by dividing the input mesh into several pieces (see Algorithm 2). We aim to divide the input surface mesh into as many evenly-sized pieces as possible. To this end, we need to find parallel planes that partition the mesh into equal-sized parts. We find such planes with the help of Principal Component Analysis (PCA). We apply PCA to the vertices of our input mesh to calculate the first principal component (PC_1). The PC_1 allows us to partition the input mesh into a maximum number of pieces. The PC_1 vector is the normal vector of these parallel planes. We then find a different point in each of these planes to define their equations. To achieve that, we project the vertices of our mesh onto the PC_1. So, we end up with a one-dimensional *projections* array, and we sort it. At this stage, we need an input parameter, the number of parts, k. With that value in hand, we create a set of indices

$$I = \{(i/k) \times |V| \;\; | \;\; i \in \{0, 1, \ldots, k-1\}\},$$

where V is the vertex set of the input and $(k-1)$ corresponds to the number of planes we need to create k pieces. For instance, when $k = 2$, I will be $I = \{|V|/2\}$, which means we get the index of the median, and by letting the plane pass through the median, we can ensure that the division is balanced. We know how to find the projected elements and back-project them to world coordinates with the indices. The resulting points, along with the PC_1, will be used to construct the planes.

Each part will contain an approximately equal number of vertices and hence an equal number of faces.

Algorithm 2 Input Partitioning

```
1: procedure INPUT_PARTITIONING(mesh, K)
2:     planes = FIND_PLANES(mesh, K)
3:     for each plane ∈ planes do
4:         INTERSECT_INSERT_POINTS(mesh, plane)
5:     end for
6:     meshes = redistribute_faces(mesh, planes)
7:                                  ▷ Distribute faces across different meshes
8:     return meshes, planes
9: end procedure
```

We need to insert new points into the mesh where the planes and mesh intersecting with these planes are known. Point insertion is essential because we want all points to be planar, simplifying the *Surface Closure* stage. The intersection and point insertion algorithm begins with iterating over all of the edges in the mesh. For each edge, we run a plane-segment intersection test. The intersection result can be a segment, a point, or nothing. We split the edge if the intersection is a point and the plane is not too close to the edge's endpoints. Splitting the edge will introduce a new point between the two endpoints of the edge. We set the new point's location as the location of the intersection point. If the intersection is a line segment, we do not split it because the line segment is on the plane, eliminating the necessity for point insertion. We also skip an edge if the plane almost intersects one of its endpoints. Omitting this step would introduce nearly duplicate points and tiny triangles that might be considered a self-intersection without sufficient floating-point precision. Figure 3.1 illustrates before and after the insertion of new points. After the intersection points are inserted into the mesh, we distribute the faces of the mesh into parts, ending up with k mesh objects. We remove those vertices during the postprocessing stage to ensure all input faces are present in the output.

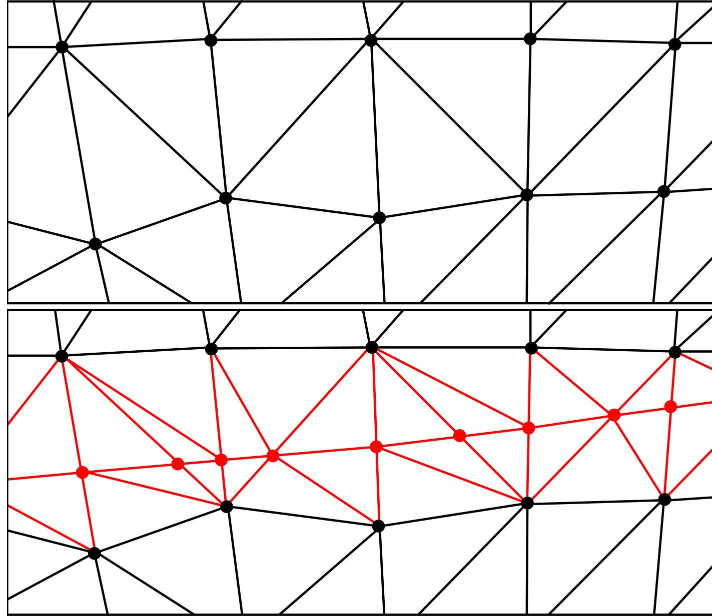

Figure 3.1: The illustration of intersection and point insertion. The top image shows the input mesh before running the algorithm. The bottom image shows the result after point insertion. Newly inserted points are shown in red.

3.3 Surface Closure

We apply a hole-filling operation on open boundaries. Although the state-of-the-art offers various methods to close a surface that nicely follows the curvature of the surface, they do not satisfy our needs [35, 36]. We aim to close the hole so that it is guaranteed to be a planar, and filling that boundary reduces to the 2D CDT problem.

Each part produced in the previous stage has open boundaries. We need to fill the holes because *TetGen* can only work on closed meshes. We use 2D Constrained Delaunay Triangulation to close the boundaries after finding them. The straightforward method of finding each boundary and triangulating the space inside might fail when the object has a genus of more than zero or a concavity in the input.

To handle all kinds of inputs, we do the following. First, we detect all boundaries and store them in an array. Each boundary will be a simple polygon, not

intersecting one another. Then, we sort the array of boundary polygons in decreasing order of the polygonal area. We also create an array to keep track of potential parent polygons. We iterate over the sorted array and check if this polygon is the child of any polygon in the parent polygons array. If this is the case, we mark this polygon as the child of the parent polygon. Otherwise, we insert that polygon into the parent polygons array. We run a 2D CDT for each parent polygon where the constraint segments are the edges of all child polygons and the polygon itself. This process can fill the holes that should not be filled. To fix that, we run a breadth-first search on the triangulation to eliminate unnecessary triangles using the breadth-first search (BFS) implementation in CGAL [37]. This BFS implementation is similar to Shewchuck's "triangle-eating virus" algorithm [38]. Algorithm 3 gives the pseudo-code of the surface closure algorithm.

After we obtain the final CDT, we further refine the triangulation to increase the quality of the triangles. The refinement stage subdivides the triangles by considering a density control factor parameter [37]. Increasing the value of this parameter leads to more uniform triangles, as illustrated in Figure 3.2. However, this process also generates many new points and triangles, complicating the object to be tetrahedralized. We then insert this triangulation to meshes on both sides of the dividing plane, reversing the triangle vertex orders before adding them to the second mesh to achieve a consistent geometry. In this way, we obtain two closed, watertight, and intersection-free meshes, just as TetGen requires (see Figure 3.3).

Figures 3.4 and 3.5 illustrate four cases with different topologies for the surface closure process. In each row, the leftmost image is the input mesh; the middle one is the bottom piece of the mesh when cut in half; the last one is after triangulating the boundaries. These four cases are as follows:

Case 1: This is the simplest case with a genus zero object.

Case 2: This is a genus one object. When we cut it in halves, we obtain two nested boundary cycles. We apply CDT using the edges of both boundaries but only keep the triangles between them eventually.

Case 3: It is similar to Case 2, but we put another object inside the hole. After cut, we have three boundary cycles inter-bedded. Again, the CDT is

Algorithm 3 Surface Closure

```
 1: procedure CLOSE_SURFACE(mesh_left, mesh_right,
                 plane, density_factor)
 2:     boundaries = extract_boundaries(mesh_left)
 3:     sort(boundaries)
 4:     parent_boundaries = []
 5:     for i = 0; i < len(boundaries); i++ do
 6:         boundary = boundaries[i]
 7:         parent_index = -1
 8:         for j = 0; j < parent_boundaries.size(); j++ do
 9:             if boundary ∈ parent_boundaries[i] then
10:                 parent_index = j
11:                 break
12:             end if
13:         end for
14:         if parent_index == -1 then
15:             parent_boundaries.add(Pair(i, [i]))
16:         else
17:             parent_boundaries[parent_index].second.add(i)
18:         end if
19:     end for
20:     # CDT Begins
21:     cdt = CDT(plane.normal)
22:     for each parent_boundary ∈ parent_boundaries do
23:         for each boundaries ∈ parent_boundary.second do
24:             cdt.add_constraints(boundaries)
25:         end for
26:     end for
27:     triangles = cdt.get_triangles()
28:     triangles = refine(triangles, density_factor)
29:     triangles = BFS(triangles)
30:     mesh_left.insert_triangles(triangles)
31:     triangles = triangles.reverse_order()
32:     mesh_right.insert_triangles(triangles)
33: end procedure
```

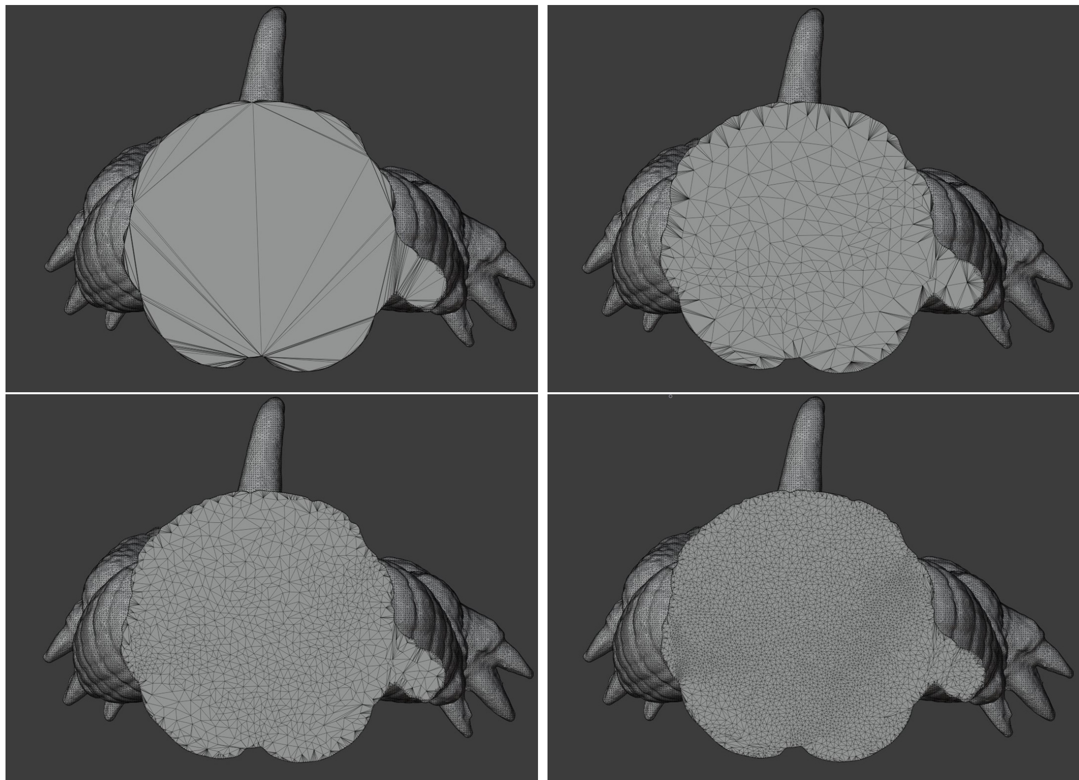

Figure 3.2: The illustration of the refinement stage on the Armadillo model. The top-left image shows the mesh without refinement (only CDT is applied). We use refinement to the others with the density factor 0.1, 0.2, and 0.4 for the top-right, bottom-left, and bottom-right images, respectively.

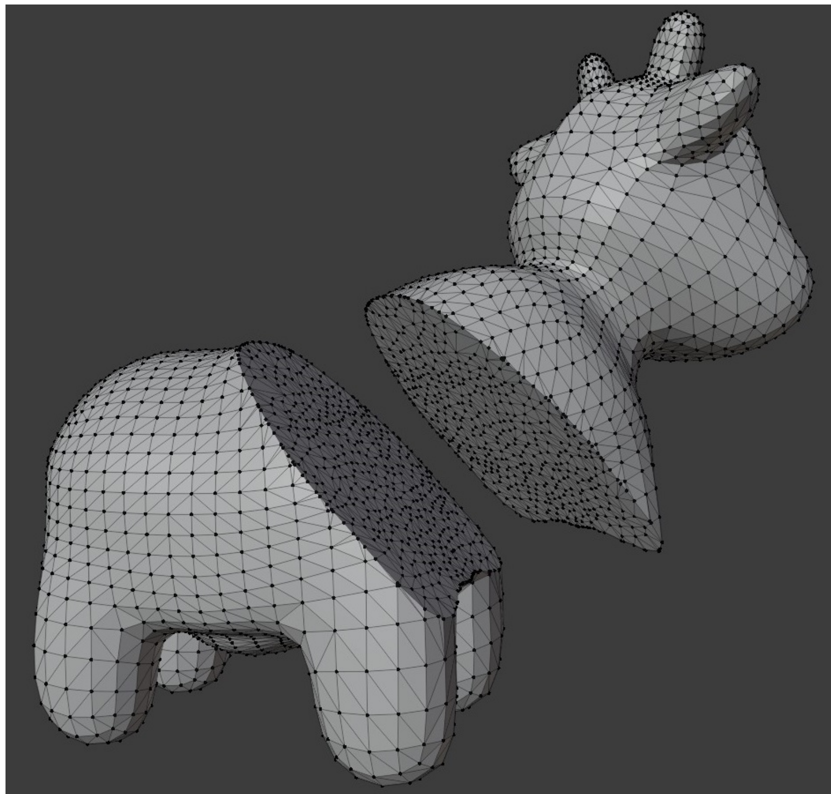

Figure 3.3: The illustration of the parts after undergoing the surface closure stage. The hole is filled using triangulation. The resulting triangulation is inserted into both left and right meshes.

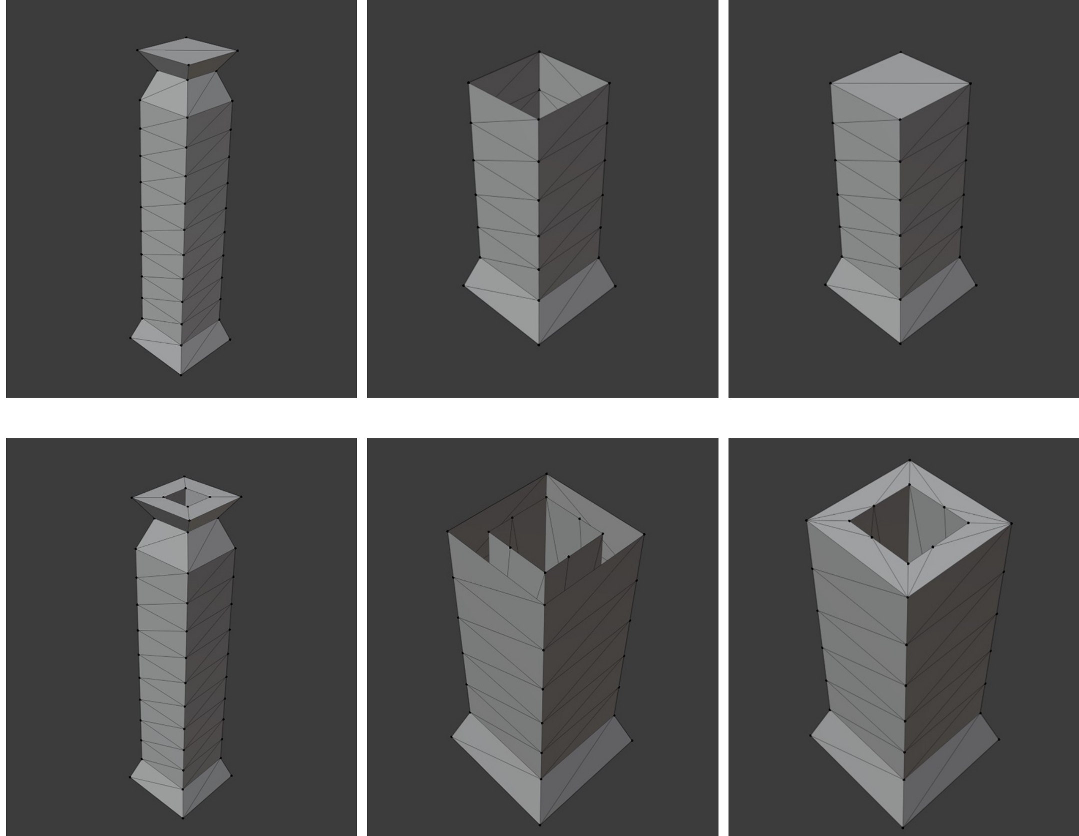

Figure 3.4: The illustration of handling inputs with various topological structures during input partitioning (first part). *First row:* the simplest case with a genus zero object (Case 1). *Second row:* a genus one object (Case 2). When we cut it in halves, we obtain two nested boundary cycles. We apply CDT using the edges of both boundaries but only keep the triangles between them eventually.

applied to all edges, but only necessary ones are kept.

Case 4: This is the combination of previous cases.

We illustrate the surface closure process for the boundary polygons shown in Figure 3.6. In this figure, the white regions in the image correspond to holes, while the purple areas correspond to our input domain. As a result of the parent finding algorithm, we can conclude that there are two-parent polygons here, shown in the black border. We then run CDT on all edges of the parent polygon and all the polygons inside it. There are three polygons inside the left parent polygon, whereas the parent polygon on the right is alone. We aim to form triangles in only purple regions. However, CDT will form triangles in the white areas as

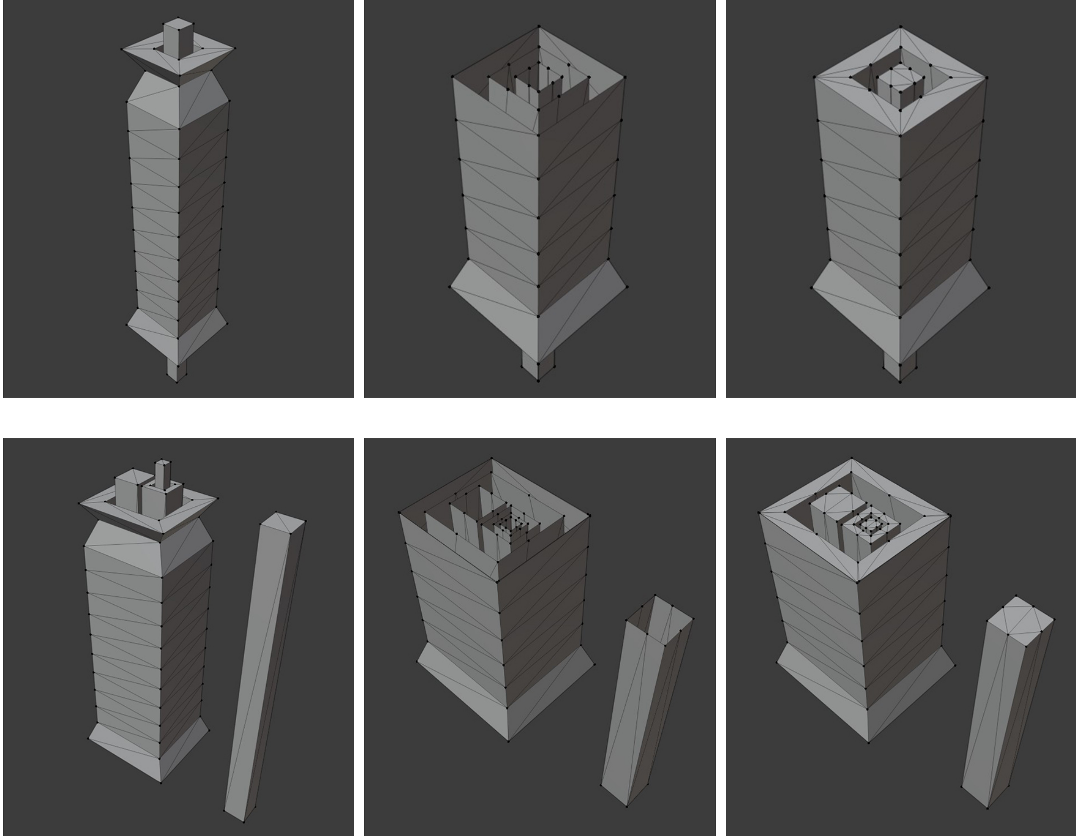

Figure 3.5: The illustration of handling inputs with various topological structures during input partitioning (second part). *First row:* similar to Case 2, but we put another object inside the hole (Case 3). After cut, we have three boundary cycles inter-bedded. Again, the CDT is applied to all edges, but only necessary ones are kept. *Second row:* the combination of the previous cases (Case 4).

well. We run a BFS algorithm to eliminate them. Figure 3.7 illustrates the input partitioning and surface closure processes for Bunny and Armadillo objects.

3.4 Merge

We merge several tetrahedral mesh (tetmesh) files in the merging stage and create one final tetmesh. One difficulty with the merge step is finding correspondence between tetrahedra around the cut region. Because each part is tetrahedralized independently, the neighboring tetrahedra at different parts will not be aware of

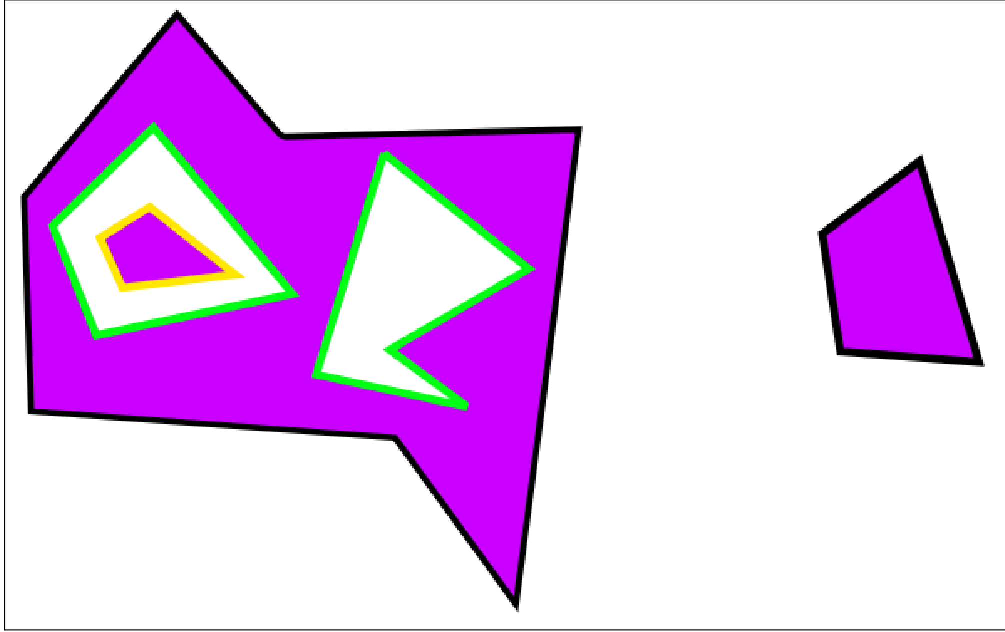

Figure 3.6: The 2D view of example boundary polygons generated after surface partitioning.

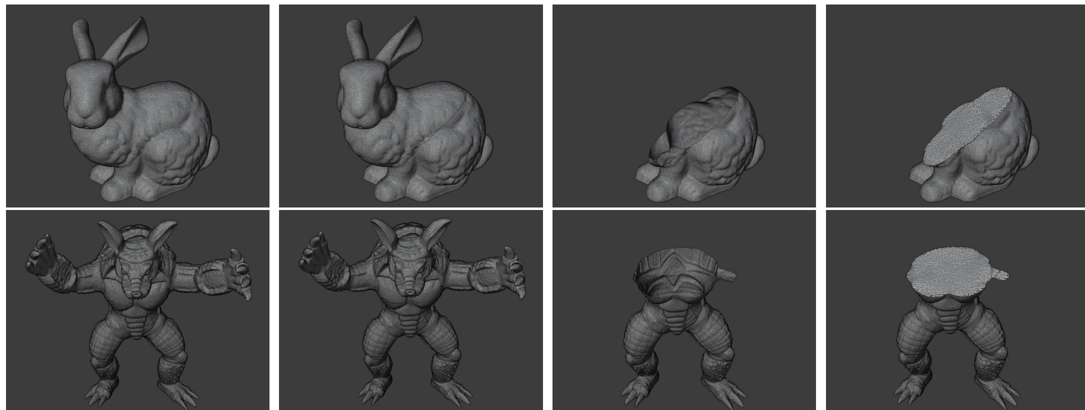

Figure 3.7: The input partitioning and surface closure stages are illustrated on Bunny (top row) and Armadillo objects (bottom row). The first column is the input mesh; the second column is the mesh after point insertion; the third column is the object's bottom half; the last column is the bottom half after the surface closure algorithm.

one another. To find these missing neighbor relations, we store the neighborhood information during the *Surface Closure* stage, as we create triangles to close the boundary and use it in the merge stage. Figure 3.8 shows tetrahedral meshes generated from the merge stage. Since we know that TetGen will preserve the triangles, eventually, the triangles around the cut region will perfectly fit after the tetrahedral mesh is created.

During the merge step, we apply postprocessing to remove vertices introduced while partitioning the input. We keep track of such vertices and remove them from the tetrahedral mesh, which would create a cavity in the tetrahedral mesh. Then, we tetrahedralize the cavity using TetGen. We ensure that the input mesh's original faces stay intact thanks to this operation. Figure 3.9 shows example tetrahedral meshes with and without the postprocessing stage and the TetGen output.

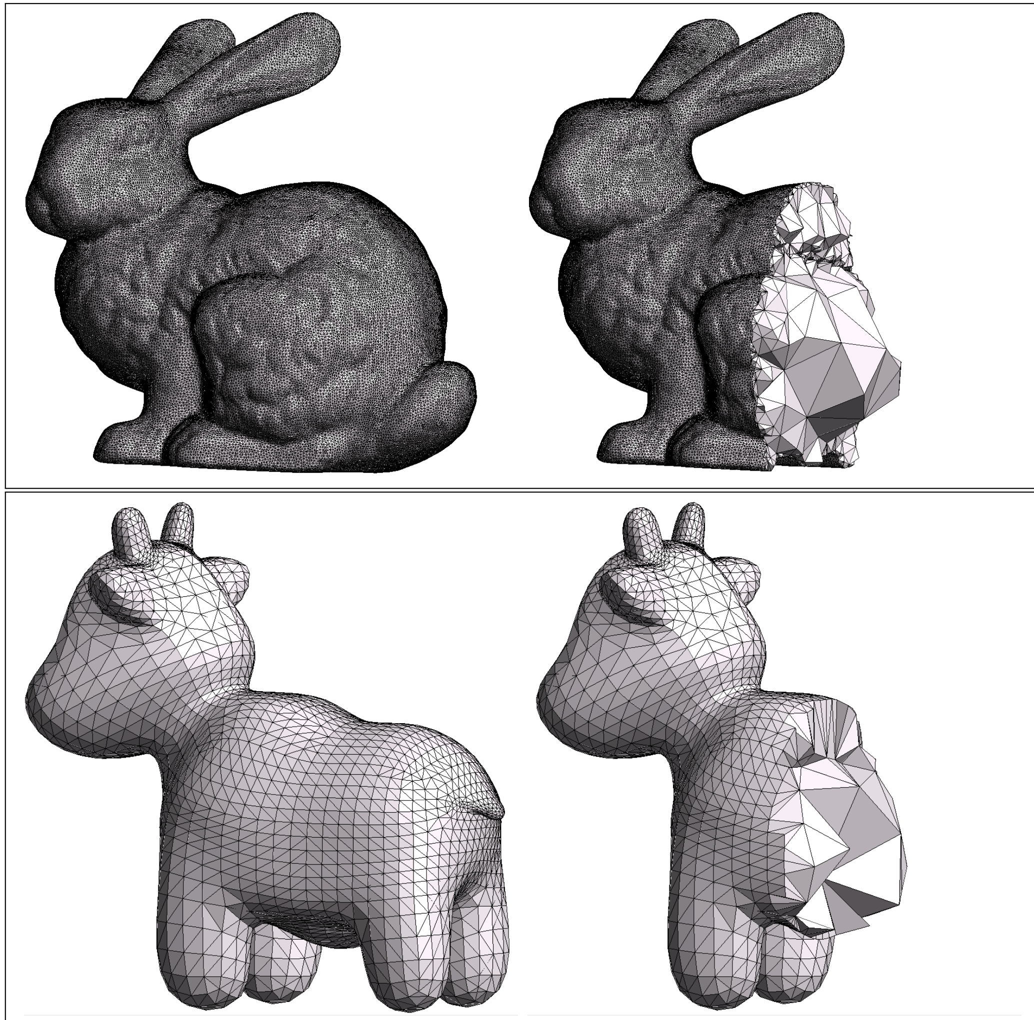

Figure 3.8: Example tetrahedral meshes generated with our implementation. Tetrahedral mesh with edges and faces (left). The cut mesh to show tetrahedra inside the model (right).

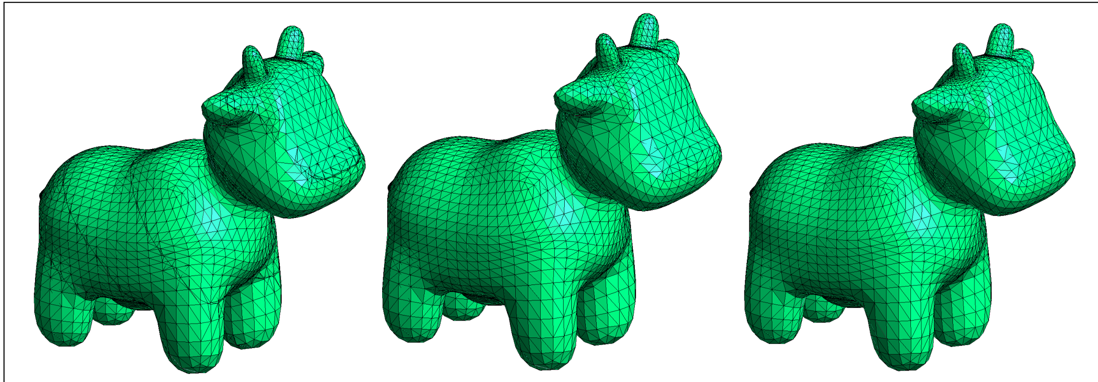

Figure 3.9: Example tetrahedral mesh outputs illustrating the result of the post-processing step. Postprocessing is enabled in the middle image but not in the left image. The right image is the result of TetGen. All extra vertices on the boundary are removed, and the constrained faces are faithful to the input mesh. Our result with postprocessing enabled is identical to TetGen's output, which is the right image.

Chapter 4

Modes of the Algorithm

Thanks to its divide-and-conquer nature, our algorithm can be used for two purposes: *Memory Requirement Reduction* and *Parallel Processing*. When the Memory Requirement Reduction mode is activated, we use single-threaded programming and intermediate files to reduce memory usage. When Parallel Processing mode is enabled, we use parallelization to speed up the process.

4.1 Memory Requirement Reduction

The benefit of reducing memory usage is two-fold. First, it allows the tetrahedralization of objects that would be impossible due to a memory shortage. Secondly, it will enable multiple objects that require high memory to be tetrahedralize simultaneously. For example, if we have a computer with 64 GB of RAM and two objects requiring 64 and 60 GB of memory, a sequential algorithm could only process either in that machine. However, our implementation can process both simultaneously by dividing each mesh at least once.

We have taken several precautions to minimize memory footprint when it comes to implementation. More specifically, we aimed to reduce the peak memory usage of our implementation. If the computer does not have enough memory

to accommodate the peak memory needed for execution, it will terminate without processing the input. For such cases, we disable multi-threading and opt for single-threaded execution. Simultaneously processing multiple pieces requires memory to be available to hold the data for all parts. Eventually, the memory footprint will be no less than *TetGen*. Instead of keeping the meshes in memory, we store the file handles. When a part is needed, we read it from the file, and when we update it, we write the changes to the corresponding file.

4.2 Parallel Processing

Because our framework allows input mesh to be divided into several pieces, we can apply multi-threaded processing. Each piece, after division, can be tetrahedralized entirely independent of each other. If we process them simultaneously, no racing condition will occur. Hence, we parallelize the for-loop at the 8th line of Algorithm 1. The mesh object is an instance of the *Surface Mesh* class belonging to the CGAL library, which states that the object is vulnerable to race conditions [37]. This vulnerability of mesh objects prevented us from parallelizing some methods due to the high costs incurred by critical sections. Moreover, we decided not to apply a multi-threading scheme to the *Merge* stage because it is already fast and includes File I/O, which needs to be synchronized, diminishing the benefits of parallelism. To parallelize code segments, we have used OpenMP 2.0 [39].

Chapter 5

Experimental Results

We have conducted experiments on mesh quality, parallel processing performance, and memory requirements. The statistics about the meshes used in the experiments are given in Table 5.1.

Table 5.1: Vertex and face counts of the objects used in the experiments. Nefertiti2 is the high resolution version of the Nefertiti model.

	# Vertices	# Faces
Spot	2,930	5,856
Bob	5,344	10,688
Blub	7,106	14,208
Bunny	72,027	144,046
Pitt Brdg	75,081	150,170
Armadillo	172,971	345,938
Nefertiti	1,009,118	2,018,232
Neptune	2,003,932	4,007,872
Nefertiti2	6,054,698	12,109,392

5.1 Mesh Quality

We performed experiments on the quality of the resulting tetrahedral meshes. We used slim energy as the quality measure, also used in TetWild [20]. The smaller

the energy is, the higher the quality is. The final quality value for a tetrahedral mesh is the average slim energy across all tetrahedra. Table 5.2 depicts the effect of the density control parameter on the mesh quality experiments. The results show that the quality improves as we increase the density parameter. After all, when we merge the partial tetrahedral meshes, the triangles we newly created for *Surface Closure* stage will be the faces of internal tetrahedra. Hence, it is expected to observe an increase in quality. Even for the same cases, we could get higher quality meshes than TetGen, thanks to the large density parameter. Hence, the non-Delaunay triangles we introduce do not seem to create many problems.

We also investigated the effect of the post-processing/vertex removal step on the tetrahedral mesh quality. As shown in Table 5.3, removing extra vertices and tetrahedralizing the cavity increases the tetrahedral mesh quality. The quality difference between our meshes and TetGen's become quite similar with post-processing enabled. Hence, although our algorithm may create some non-Delaunay triangles, the quality difference appears slim compared to the gain achieved by reducing the memory footprint and computation time.

Table 5.2: Experiments on tetrahedral mesh quality. The quality metric is the average slim energy. The first column is the results for the standalone TetGen execution. Other columns show the values for the corresponding density control parameter. The smallest value at each row is in bold.

		Density control parameter				
Model	TetGen	0.1	0.2	0.4	0.8	1.6
Spot	6.26	7.26	6.72	**6.15**	6.23	6.33
Bob	**5.99**	7.74	6.82	6.41	6.16	6.27
Blub	7.84	8.79	8.25	7.89	7.74	**7.46**
Pitt Brdg	7.27	7.72	7.36	7.26	7.317	**7.14**
Armadillo	6.65	6.82	6.76	6.71	6.681	**6.61**

5.2 Parallel Processing

We conducted experiments to see how our parallelization scheme performs. We used a PC having two eight-core processors, equivalent to 16-core processor power.

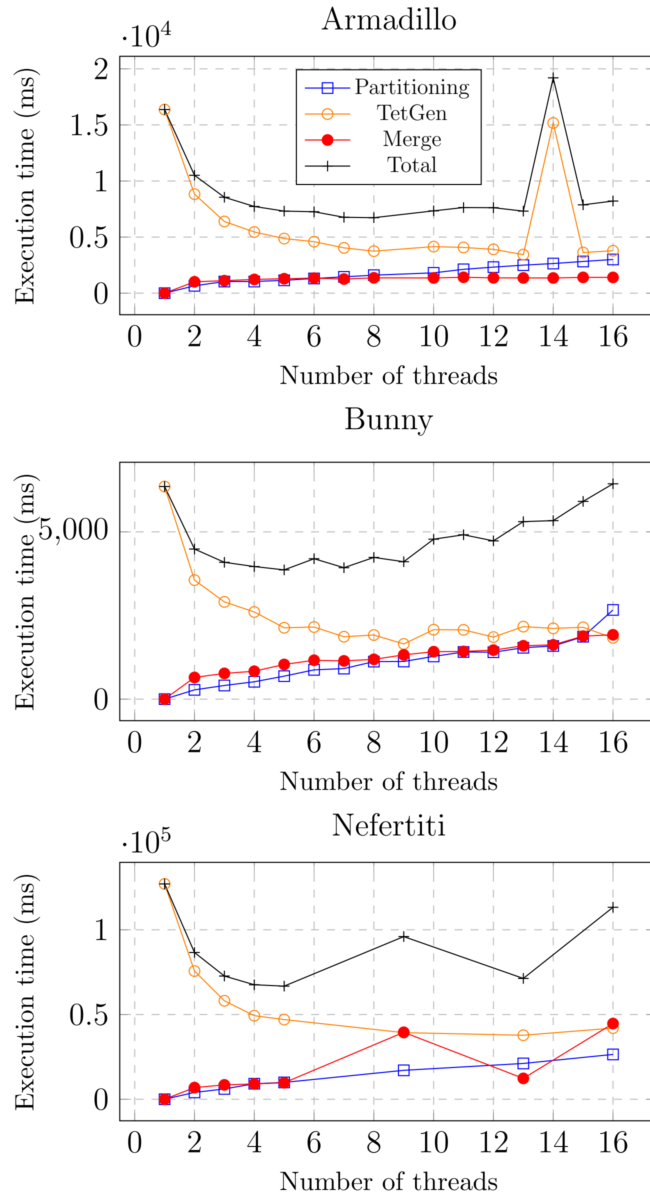

Figure 5.1: Execution time dissection for Armadillo, Bunny, and Nefertiti objects. Single-threaded execution corresponds to sequential *TetGen*.

Table 5.3: Experiments on the effects of post-processing (i.e., vertex removal) step on tetrahedral mesh quality. The quality metric is average slim energy; the lower it is, the better. The density control parameter is selected as 0.1.

Model	TetGen	without postprocessing	with postprocessing
Spot	6.26	9.59	7.26
Bob	5.99	9.91	7.74
Blub	7.84	10.84	8.79
Pitt Brdg	7.27	8.21	7.72
Armadillo	6.65	7.09	6.82

The graphs show that TetGen execution time increases as the number of threads increases. The reason may be the imbalanced data partition. If the processing of the whole mesh by TetGen on a single thread takes T time, each thread should ideally complete the execution in T/X time, where X is the number of threads. However, this ideal case does not always occur. Some threads run faster and some slower due to imbalanced input partitioning. We ensure that each part has an equal number of vertices and faces, but the execution time of each thread might be different due to topological differences. In addition, we are inserting new faces during the *Surface Closure* stage, and each part might get a different number of vertices and faces appended to it depending on the boundary polygons. All these factors lead to data imbalance. As we increase the number of pieces, this issue becomes crucial, slowing down the process. Moreover, the spike at 14 threads for Armadillo input shows that sometimes the partitioning might be imbalanced unluckily, and we may end up with a sudden increase in the execution time.

The choice of the density control parameter used in the refinement stage affects the execution time of our algorithm. Incrementing that value increases the number of triangles used to fill the holes and the quality of the triangles (cf. Section 5.1). Higher quality triangles require TetGen spend less time doing optimization on the mesh. Table 5.4 shows that the value of 0.4 seems reasonable for this parameter considering the trade-off between the computational cost and mesh quality.

Some of the cases failed because of the precision issues that occurred while inserting new points to the mesh. We used an inexact construction kernel of the CGAL [37]; this is why such failures might occur. In addition, TetGen rarely inserts Steiner points on the input triangle, preventing the merge process from running correctly. The merge fails if the triangles at both sides do not match due to new Steiner points. We count this as a failure case too.

Table 5.4: The effect of density control parameter on the execution time. The execution time is in milliseconds.

		Density control parameter				
	TetGen	0.1	0.2	0.4	0.8	1.6
Spot	120	139	115	**108**	123	174
Bob	283	475	214	225	**212**	354
Blub	413	307	**289**	318	338	467
Pitt Brdg	4502	3642	3557	**3309**	3533	3822
Armadillo	14793	**9203**	9542	9484	11131	22492

Table 5.5: Memory usage (in MB) and processing times (in seconds) for various models with post-processing enabled. When the part count is one, *TetGen* is directly used. Our implementation failed for some input-part count pairs due to the floating-point errors; these are shown with "-". The inputs that *TetGen* could not process are marked with an "x". Nefertiti2 is the high-resolution version of the Nefertiti model.

	Part counts									
	1		2		4		8		16	
Model	Mem.	Time	Mem.	Time	Mem.	Time	Mem.	Time	Mem.	Time
Armadillo	923	12	542	22	336	34	268	60	360	109
Nefertiti	5200	85	3700	166	2500	248	2200	475	3000	726
Neptune	11800	250	6700	344	4400	500	-	-	-	-
Nefertiti2	x	x	23500	1552	-	-	-	-	-	-

5.3 Memory Requirements

We tested our algorithm with several objects and observed the peak memory usage and execution times by enabling the memory reduction mode of our algorithm. We limited the available memory to 36 GB (16 GB Physical RAM + 20 GB

Table 5.6: The effect of density control parameter on the memory usage. The memory usage is in Megabytes.

		Density control parameter				
	TetGen	0.1	0.2	0.4	0.8	1.6
Spot	21	16	16	14	16	20
Bob	33	25	23	24	25	30
Blub	43	30	28	28	36	39
Pitt Brdg	388	227	228	256	237	262
Armadillo	930	547	582	565	623	944

Virtual Memory). Table 5.5 shows the results for various input-part count pairs. We excluded reading and writing times from the execution time.

As we increase the number of pieces, we see a significant decrease in peak memory usage despite increasing execution time. Moreover, TetGen could not process the "Nefertiti2" model due to high memory usage, whereas our method could tetrahedralize it. The execution time of the memory-efficient version of our algorithm appears to be reasonable compared to the original TetGen. Still, there is a continuous increase in the execution time as we increase the number of parts. Our profiling analysis shows that it should be caused by the file I/O operations that we do the save intermediate files to save the memory consumption. For the Armadillo object, approximately 75 seconds is spent during intermediate file I/O with 16 parts, corresponding to a significant portion of the execution time.

We also investigated the relation between memory usage and the density control parameter. Table 5.6 shows that increasing the density parameter increases memory usage. Although it is often less than the standalone TetGen execution, the memory usage can even exceed that in some cases if the density control parameter is too high. We expect this result because the density parameter controls the number of triangles, and its increase leads to more triangles. Creating more triangles means tetrahedralizing objects with more vertex and triangles counts, which increases memory usage.

Chapter 6

Alternative Approach

6.1 Overview

To achieve Constrained Tetrahedralization, we need to ensure that input triangles stay intact in the tetrahedral output mesh and should be at the boundary of the mesh. Our approach involves inserting points during the input partitioning stage to simplify the surface closure operation. That is, 2D triangulation works smoothly with plane surfaces. As a final step, we remove those extra vertices to ensure the triangles are preserved. An alternative to that approach is trying to do input partitioning without inserting new vertices. In this method, we partition the objects using planes again, but this time instead of inserting new points, we divide the input triangle set into disjoint sets. To close the open surfaces, we project the boundaries into the 2D plane, apply convex decomposition, put a vertex at the center of each convex sub-polygon and connect it to the rest of the vertices in the convex sub-polygon. We then project the triangulation back to 3D. Since this stage may introduce self-intersecting triangles, we locate and fix them during the repairing stage. Finally, we merge all of the pieces.

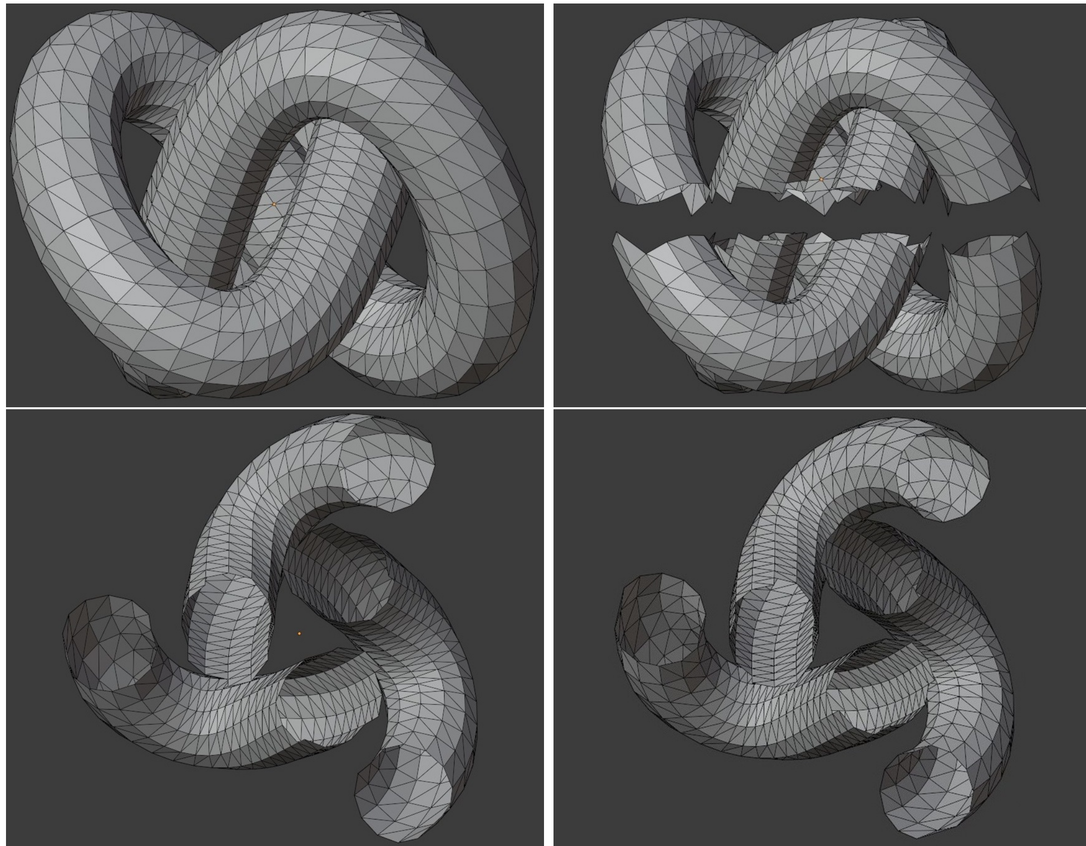

Figure 6.1: Illustration of partitioning algorithm applied on Torus Knot object (top-left). The top-right image shows the two pieces together after the partitioning. In the bottom-left and bottom-right are the cross-section views of the left and right pieces, respectively.

6.2 Partitioning Stage

The partitioning algorithm accepts an input mesh file and a 2D Plane as input and outputs the left and right mesh files. The mesh file corresponds to the name of that file on the disk. We separate the input mesh into two pieces: we put the faces whose all vertices are to the left of the dividing plane to the left piece and the rest of the faces to the right piece. This process naturally creates two meshes with open boundaries. In the next stage, we close these open boundaries. Figure 6.1 depicts an example partitioning.

Algorithm 4 Partitioning Procedure

```
procedure PARTITIONING(input_mesh, plane)
    left_mesh = {face f ∈ input_mesh |
              ∀ vertex v ∈ f, v is to the left of the plane}
    right_mesh = input_mesh - left_mesh
    return left_mesh, right_mesh
end procedure
```

6.3 Surface Closure Stage

At this stage, we have two meshes with open surfaces. Because TetGen can run constrained tetrahedralization on closed meshes, we need to close the open surfaces. To this end, we have devised a 2D-convex-decomposition-based algorithm. Our surface closure algorithm does not add a new point on the surface and is faithful to existing mesh geometry.

In our surface closure algorithm, we mainly operate on the left piece where we find a 2D triangulation, through 2D convex decomposition, that closes the left surface, and we copy this triangulation to the right so that it is closed as well. We need the same triangulation on both sides because, eventually, the tetrahedra at each piece's boundaries must strictly match for the tetrahedral mesh to be valid.

The algorithm begins with finding the polygons at the boundary. Figure 6.2 shows the boundary polygons in orange color. There may be several polygons; we capture and process them one by one. After we extract the polygons, we project them on the 2D plane, dividing the object. We then end up with a 2D polygon for each boundary. Now, we run the steps illustrated in Figure 6.3 for each of them. First, we call a convex decomposition procedure to generate convex sub-polygons out of the projected polygon. The decomposition need not be optimal; approximate solutions are acceptable for our algorithm. With the sub-polygons in hand, we can insert new points at their centroid and then connect this centroid with all the other vertices in the sub-polygon. One should note that we insert the new points inside the object, not on the surface. There may be a degenerate case where the sub-polygon is a triangle. We handle these cases during the next -Repairing- stage. Overall, this process has the effect of closing the surface through

triangulation. We, finally, copy this triangulation to the right piece so that it is closed too. Figure 6.4 shows the result of applying that algorithm to the left mesh in Figure 6.2. The right piece also has the same triangulation pasted on it.

Although this stage seems to close the surface, it is prone to generating self-intersection problems. We shall discuss our approach to solving them in the next section.

Algorithm 5 Surface Closure Algorithm

```
1: procedure CLOSE_SURFACES(left_vertices, left_triangles,
        right_vertices, right_triangles, plane)
2:     left_boundary = FIND_BOUNDARY(left_vertices, left_triangles)
3:     left_boundary_proj = PLANE_PROJECTION(left_boundary, plane)
4:     decomp = 2D_CONVEX_DECOMP(left_boundary_proj)
5:     left_vertices, left_triangles =
           ADD_NEW_TRIANGLES(decomp, left_vertices, left_triangles)
6:     COPY_TRIANGULATION(left_vertices, left_triangles,
           right_vertices, right_triangles)
7:     return left_vertices, left_triangles, right_vertices, right_triangles
8: end procedure
```

6.4 Repairing Stage

Our repairing stage consists of two steps: finding intersections and resolving intersections. To find the intersections, we run TetGen in diagnostic mode and feed the left and right pieces into it. TetGen locates and reports the intersecting faces in the left and right parts closed by our surface closure algorithm. We receive the intersecting faces from TetGen and run our repairing algorithm to attempt to resolve the problems. TetGen's diagnostic mode tries to find intersections over the whole object, but we know the intersections will be around the separation region. Using this information, we modified TetGen's diagnosis procedure to only check the triangles around the cut plane for an intersection. We dictate TetGen to only look at triangles whose vertices are at most D distance away from the separation plane. This restriction provides a significant speed-up. We calculate D as $D = a + b$ where a is the longest edge in the input mesh, and b is the

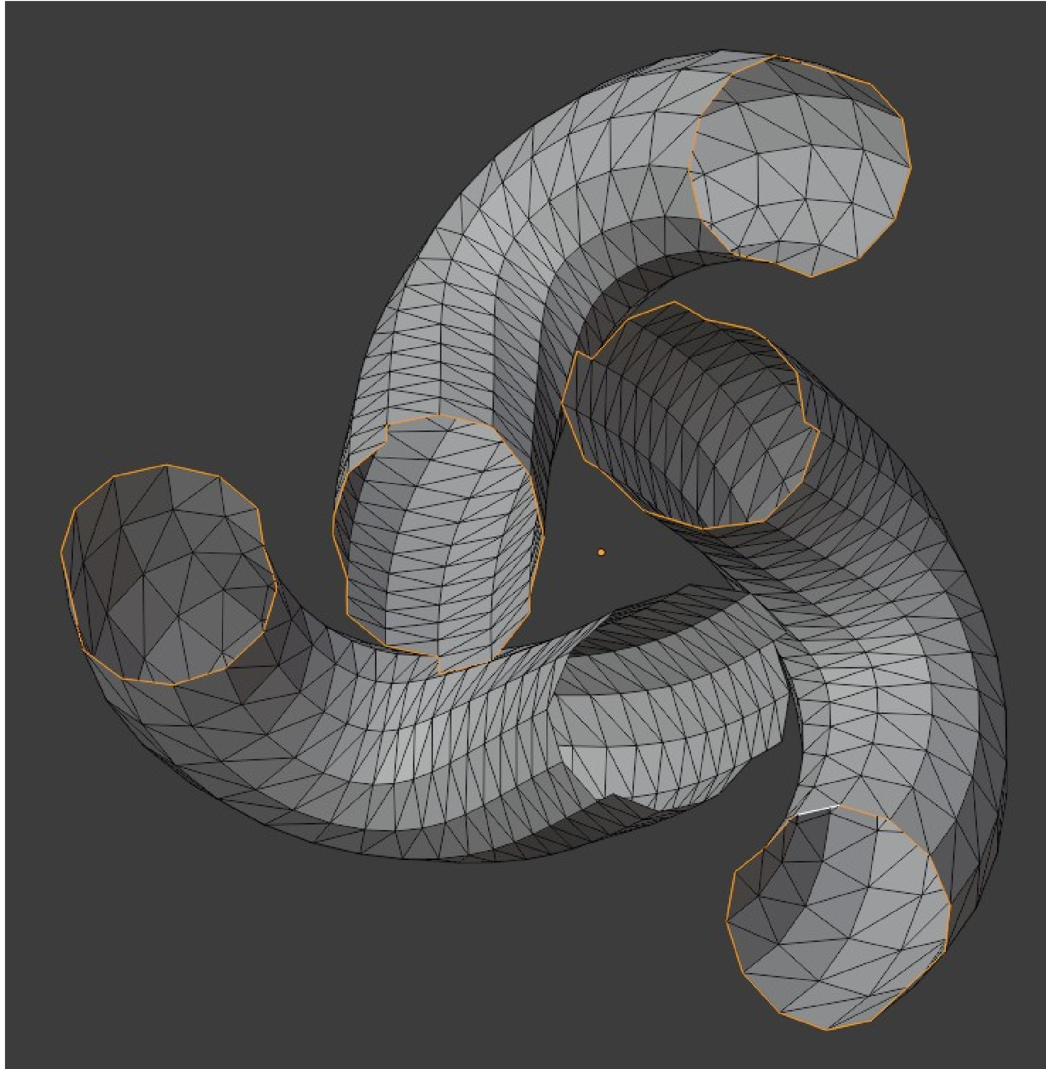

Figure 6.2: Illustration of the left mesh with its boundary polygons marked in orange.

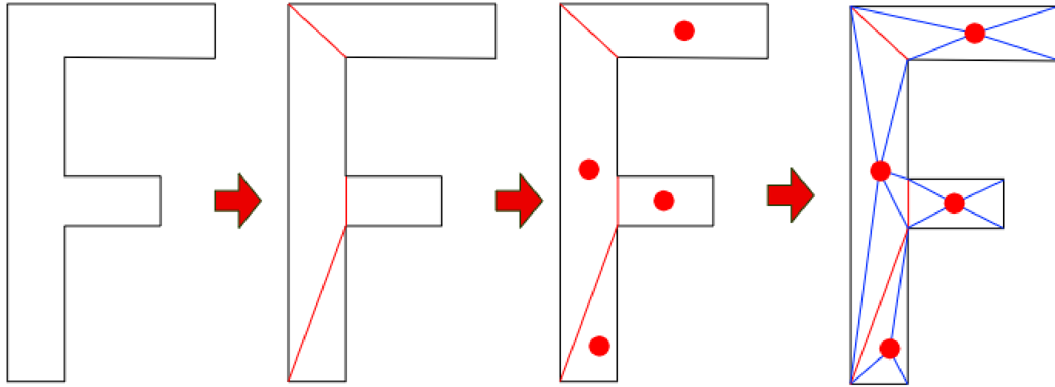

Figure 6.3: Illustration of the surface closure algorithm: from left to right: projected boundary polygon; Planar Straight Line Graph (PSLG) after convex decomposition where red edges are newly added; centroids calculated for each convex polygon; and sub-polygon vertices and centroids of each convex polygon connected with blue edges.

distance between the plane and the farthest boundary point from that plane. We know the intersections generated by our Surface Closure algorithm will only affect triangles that have vertices on the boundary. Therefore, all triangle points with self-intersection will be at most D away from the plane. Figure 6.5 illustrates the a and b values where the longest edge in the input mesh is assumed to be near the boundary.

With the self-intersecting faces located, we can move on to repair the problematic regions. The edge-face intersection is a major problem that emerges through the surface closure stage. As illustrated in Figure 6.7 (a), ABCDEF region contains such a problematic case. The BF edge in the graph intersects with the faces of the input mesh. The shape created by the vertices B, C, D, E, and F contains some edge-face intersections. We remove all edges and vertices formed by our surface closure procedure within the ABCDEF region to remedy this problem. We have a cavity where ABCDEF is the boundary polygon as in Figure 6.7 (b). After that, we connect vertex A with every other vertex in the polygon to effectively form the triangles ABC, ACD, ADE, and AEF. Figure 6.7 (c) depicts the final configuration of the triangles.

We also handle degenerate cases where a convex sub-polygon is a triangle.

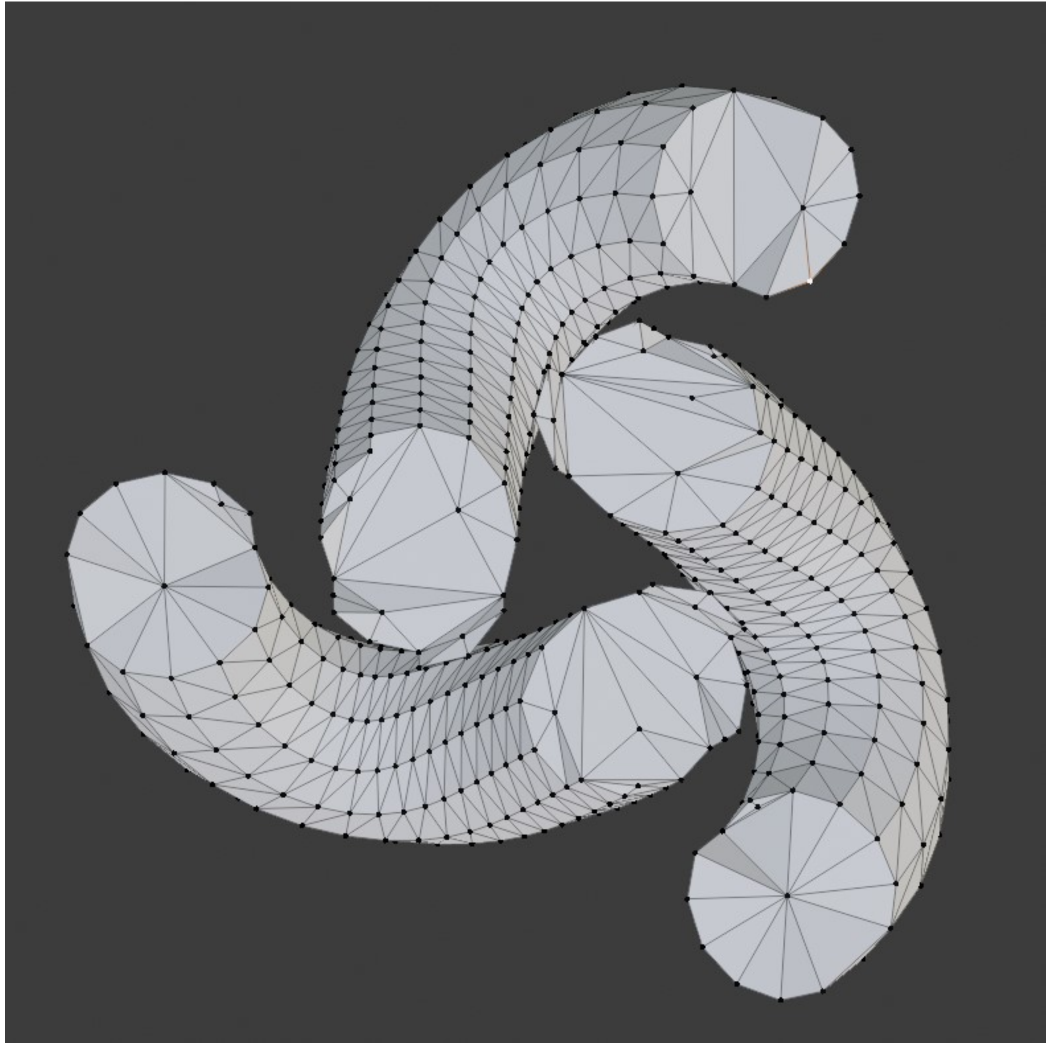

Figure 6.4: Illustration of the left mesh with its boundaries closed.

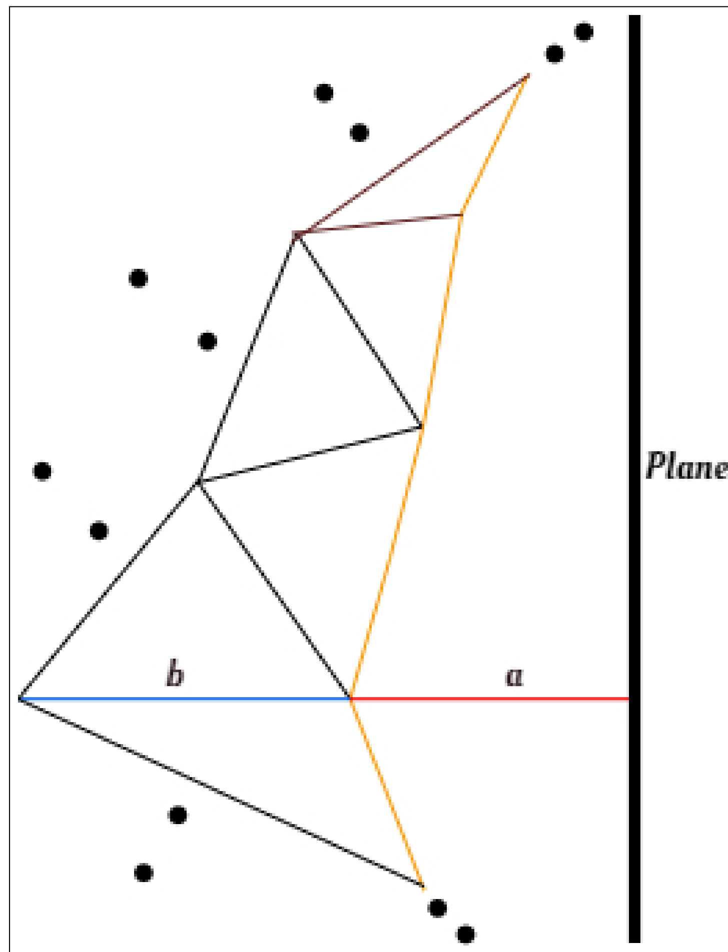

Figure 6.5: Illustration of the TetGen's diagnosis speed-up process: orange line is a portion of the boundary, the red line corresponds to a value, whereas the blue line to b.

Figure 6.6 illustrates such a case. This case is problematic because it introduces a vertex on the triangle that causes vertex-face intersection. Our approach here is to connect all the vertices to a previously calculated centroid point K, which is not visible in the figure. We calculate point K by finding the vertex connected to both A and C. We also remove the point at the centroid of triangle ABC. Even if we did not process that case, TetGen could still process the mesh. Nevertheless, we handled that, to be sure.

This repairing procedure works under some assumptions, which we shall discuss in the Limitations section.

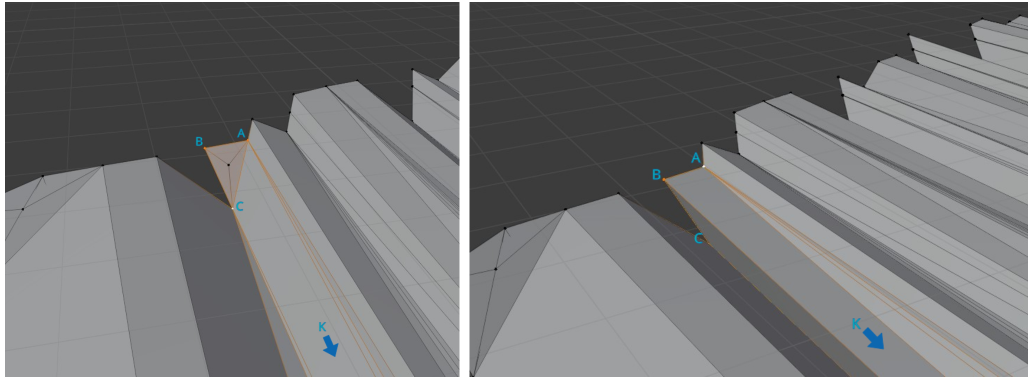

Figure 6.6: Handling the degenerate case. A, B, and C are the vertices of the triangle. (a) before handling. (c) after handling.

Algorithm 6 Repairing Algorithm

1: **procedure** REPAIR(*left_vertices, left_triangles, right_vertices, right_triangles*)
2:     *intersections* = TETGEN_DIAGNOSE(*left_vertices, left_triangles, right_vertices, right_triangles*)
3:     **for each** *polygon* ∈ *intersections* **do**
4:         REMOVE_VERTICES_EDGES(*left_vertices, left_triangles, right_vertices, right_triangles, polygon*)
5:         ADD_NEW_TRIANGLES(*left_vertices, left_triangles, right_vertices, right_triangles, polygon*)
6:     **end for**
7:     **return** *left_vertices, left_triangles, right_vertices, right_triangles*
8: **end procedure**

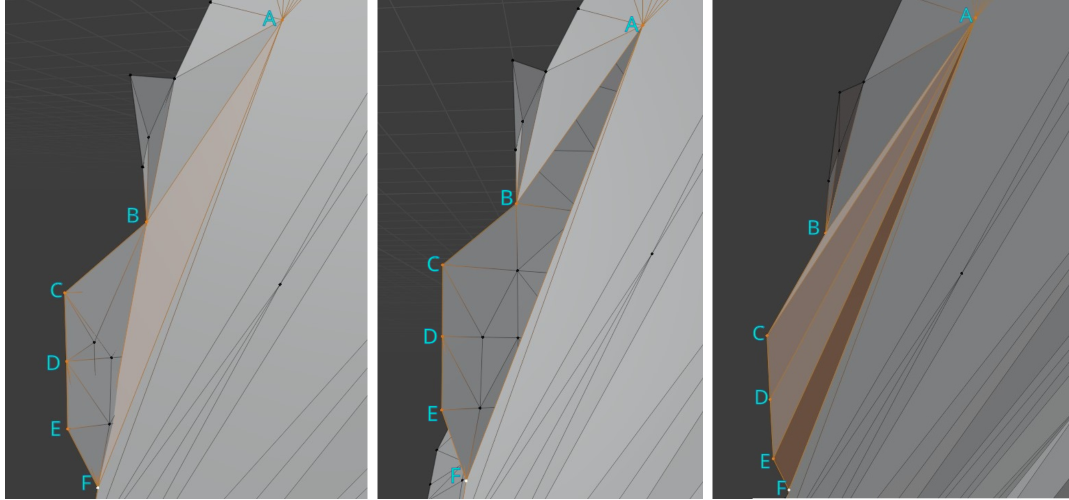

Figure 6.7: Handling edge-face intersections. (a) the problematic case where the edge BF intersects with the object. (b) the problematic edges and vertices are removed. (c) new triangles are created.

6.5 Limitations of the Alternative Approach

Although this approach removes the need to insert extra vertices, it comes with its limitations, which prevented us from using it in the first place. The biggest problem with this approach is that it relies too much on floating-point precision. The repairing stage depends on TetGen's intersection test to find and repair problematic cases. However, that stage is not guaranteed always to work correctly. In addition, the repairing stage itself can insert new self-intersecting triangles. All of these are caused by the fact that we project a polyline in 3D into 2D space, which cannot guarantee that the non-self-intersecting triangles created in 2D will not self-intersect in 3D. As a result, this approach cannot always succeed, so we did not use it in the proposed algorithm.

Chapter 7

Conclusion

We propose a divide-and-conquer algorithm that can be used to reduce memory usage or speed up the constrained tetrahedral meshing process. Although our algorithm may introduce some non-Delaunay triangles, it can increase the quality of the tetrahedral mesh. Despite non-Delaunay triangles, the increase in quality makes our method useful. We can even successfully tetrahedralize meshes that TetGen cannot do due to lack of memory. Although our input partitioning stage introduces new vertices, we remove them during merge to conserve the input triangles.

We also experimented with an alternative approach that does not require the creation of a middle region that must be tetrahedralized. However, it comes with deficiencies, such as the 2D projection operation necessary for the surface closure stage is not guaranteed to be bijective and unreliable self-intersection tests required after triangulating the 2D surface to close the 2D surface. Besides, near-degenerate configurations may easily make it fail due to rounding errors. Hence, we did not adopt it in our proposed divide-and-conquer CDT algorithm.

We could extend the proposed divide-and-conquer 3D CDT algorithm in various ways. Firstly, we used PCA and parallel planes during input partitioning to reduce the overhead. To set a trade-off between speed and more balanced

decomposition, we could experiment with other approaches, such as convex decomposition and recursive PCA. As convex decomposition is relatively slow, and partitioning with non-parallel planes -as with recursive PCA- requires a complicated merge step (i.e., BSP trees to keep track of neighboring pieces efficiently), the overall process may be slower, but decomposition could be more balanced. Secondly, we applied our framework to only TetGen. It can, however, be used with other meshing tools such as TetWild. TetWild may also suffer from insufficient memory, so it would benefit from such an approach.

Currently, we manually determine the value of the density control parameter. We would automatically determine that parameter based on the average area of triangles in the mesh. That way, the mesh density would be similar to the triangular input mesh, and we would prevent imbalance in the output where triangles in the surface closure are more or less dense than the input.

Bibliography

[1] T. I. Šušteršič and N. Filipovic, "Computational modeling of dry-powder inhalers for pulmonary drug delivery," in *Computational Modeling in Bioengineering and Bioinformatics* (N. Filipovic, ed.), pp. 257–288, Academic Press, 2020.

[2] P. Anderson, S. Fels, N. M. Harandi, A. Ho, S. Moisik, C. A. Sánchez, I. Stavness, and K. Tang, "FRANK: a hybrid 3D biomechanical model of the head and neck," in *Biomechanics of Living Organs* (Y. Payan and J. Ohayon, eds.), vol. 1 of *Translational Epigenetics*, pp. 413–447, Oxford: Academic Press, 2017.

[3] Y. Payan and J. Ohayon, eds., *Biomechanics of Living Organs*, vol. 1 of *Translational Epigenetics*. Oxford: Academic Press, 2017.

[4] J. Tu, G.-H. Yeoh, and C. Liu, "CFD mesh generation: A practical guideline," in *Computational Fluid Dynamics* (J. Tu, G.-H. Yeoh, and C. Liu, eds.), pp. 125–154, Butterworth-Heinemann, third edition ed., 2018.

[5] N. Molino, R. Bridson, J. Teran, and R. Fedkiw, "A crystalline, red green strategy for meshing highly deformable objects with tetrahedra," in *Proceedings of the 12th International Meshing Roundtable, IMR 2003* (J. Shepherd, ed.), pp. 103–114, 2003.

[6] J. Teran, E. Sifakis, G. Irving, and R. Fedkiw, "Robust quasistatic finite elements and flesh simulation," in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, (New York, NY, USA), p. 181–190, Association for Computing Machinery, 2005.

[7] E. Sifakis, K. G. Der, and R. Fedkiw, "Arbitrary cutting of deformable tetrahedralized objects," in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, (Goslar, DEU), p. 73–80, Eurographics Association, 2007.

[8] N. Montazerin, G. Akbari, and M. Mahmoodi, *Developments in Turbomachinery Flow: Forward Curved Centrifugal Fans*, vol. 1. Sawston, U.K.: Woodhead Publishing, 2015.

[9] M. Driscoll, "The impact of the finite element method on medical device design," *Journal of Medical and Biological Engineering*, vol. 39, p. 171–172, 2019.

[10] F. Mollica and L. Ambrosio, "The finite element method for the design of biomedical devices," in *Biomaterials in Hand Surgery* (A. Merolli and T. J. Joyce, eds.), pp. 31–45, Milano: Springer Milan, 2009.

[11] A. Wittek and K. Miller, "Computational biomechanics for medical image analysis," in *Handbook of Medical Image Computing and Computer Assisted Intervention* (S. K. Zhou, D. Rueckert, and G. Fichtinger, eds.), The Elsevier and MICCAI Society Book Series, pp. 953–977, Academic Press, 2020.

[12] M. Freutel, H. Schmidt, L. Dürselen, A. Ignatius, and F. Galbusera, "Finite element modeling of soft tissues: Material models, tissue interaction and challenges," *Clinical Biomechanics*, vol. 29, no. 4, pp. 363–372, 2014.

[13] F. Galbusera and F. Niemeyer, "Mathematical and finite element modeling," in *Biomechanics of the Spine* (F. Galbusera and H.-J. Wilke, eds.), pp. 239–255, Academic Press, 2018.

[14] T. Schneider, Y. Hu, X. Gao, J. Dumas, D. Zorin, and D. Panozzo, "A large scale comparison of tetrahedral and hexahedral elements for finite element analysis," *arXiv preprint arXiv:1903.09332*, 2019.

[15] Z. Jiang, Z. Zhang, Y. Hu, T. Schneider, D. Zorin, and D. Panozzo, "Bijective and coarse high-order tetrahedral meshes," *ACM Transactions on Graphics (TOG)*, vol. 40, no. 4, pp. 1–16, 2021.

[16] M. Alexa, "Harmonic triangulations," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–14, 2019.

[17] X.-Y. Li and S.-H. Teng, "Generating well-shaped Delaunay meshes in 3D," in *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, (USA), p. 28–37, Society for Industrial and Applied Mathematics, 2001.

[18] C. Hazlewood, "Approximating constrained tetrahedrizations," *Computer Aided Geometric Design*, vol. 10, no. 1, pp. 67–87, 1993.

[19] H. Si, "TetGen, a Delaunay-based Quality Tetrahedral Mesh Generator," *ACM Transactions on Mathematical Software*, vol. 41, no. 2, pp. 1–36, 2015.

[20] Y. Hu, Q. Zhou, X. Gao, A. Jacobson, D. Zorin, and D. Panozzo, "Tetrahedral meshing in the wild," *ACM Trans. Graph.*, vol. 37, pp. 60:1–60:14, July 2018.

[21] Y. Hu, T. Schneider, B. Wang, D. Zorin, and D. Panozzo, "Fast tetrahedral meshing in the wild," *ACM Trans. Graph.*, vol. 39, pp. 117:1–117:18, July 2020.

[22] L. P. Chew, "Guaranteed-quality mesh generation for curved surfaces," in *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, (New York, NY, USA), p. 274–280, Association for Computing Machinery, 1993.

[23] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*. Cambridge, MA, USA: MIT Press, 2nd. (revised) ed., 1998.

[24] A. N. Chernikov and N. P. Chrisochoides, "Algorithm 872: Parallel 2d constrained delaunay mesh generation," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 1, pp. 1–20, 2008.

[25] L. Linardakis and N. Chrisochoides, "Algorithm 870: A static geometric medial axis domain decomposition in 2D Euclidean space," *ACM Transactions on Mathematical Software*, vol. 34, no. 1, pp. 1–28, 2008.

[26] N. Coll and M. Guerrieri, "Parallel constrained Delaunay triangulation on the GPU," *International Journal of Geographical Information Science*, vol. 31, no. 7, pp. 1467–1484, 2017.

[27] D. K. Blandford, G. E. Blelloch, and C. Kadow, "Engineering a compact parallel Delaunay algorithm in 3D," in *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SoCG '06, pp. 292–300, 2006.

[28] A. N. Chernikov and N. P. Chrisochoides, "Three-dimensional Delaunay refinement for multi-core processors," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, (New York, NY, USA), p. 214–224, Association for Computing Machinery, 2008.

[29] P. Cignoni, C. Montani, and R. Scopigno, "DeWall: A Fast Divide and Conquer Delaunay Triangulation Algorithm in E^d," *Computer-Aided Design*, vol. 30, no. 5, pp. 333–341, 1998.

[30] M.-B. Chen, T.-R. Chuang, and J.-J. Wu, "Efficient parallel implementations of near Delaunay triangulation with high performance Fortran," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 12, pp. 1143–1159, 2004.

[31] C. Marot, J. Pellerin, and J. Remacle, "One machine, one minute, three billion tetrahedra," *International Journal for Numerical Methods in Engineering*, vol. 117, pp. 967–990, Dec 2019.

[32] B. J. Joshi and S. Ourselin, "BSP-assisted constrained tetrahedralization," in *Proceedings of the 12th International Meshing Roundtable (IMR)*, pp. 251–260, 2003.

[33] M. Smolik and V. Skala, "Fast parallel triangulation algorithm of large data sets in E^2 and E^3 for in-core and out-core memory processing," in *Proceedings of the International Conference on Computational Science and Its Applications*, ICCSA '14, pp. 301–314, Springer, 2014.

[34] Z. Erkoç, A. Aman, U. Güdükbay, and H. Si, "Out-of-core constrained delaunay tetrahedralizations for large scenes," in *Numerical Geometry, Grid Generation and Scientific Computing* (V. A. Garanzha, L. Kamenski, and H. Si, eds.), (Cham), pp. 113–124, Springer International Publishing, 2021.

[35] W. Zhao, S. Gao, and H. Lin, "A robust hole-filling algorithm for triangular mesh," *The Visual Computer*, vol. 23, no. 12, pp. 987–997, 2007.

[36] L. S. Tekumalla and E. Cohen, "A hole-filling algorithm for triangular meshes," *School of Computing, University of Utah, UUCS-04-019, UT, USA*, vol. 2, 2004.

[37] The CGAL Project, *CGAL User and Reference Manual*, 5.0.2 ed., 2020. Available at `https://doc.cgal.org/5.0.2/Manual/packages.html`, Accessed at 14 June 2022.

[38] J. R. Shewchuk, "Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator," in *Workshop on Applied Computational Geometry*, pp. 203–222, Springer, 1996.

[39] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science Engineering*, vol. 5, no. 1, pp. 46–55, 1998.