Winter 12-1-2013

An Algorithm for Triangulating 3D Polygons

Ming Zou
Washington University in St. Louis

Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Tao Ju, Chair
Robert Pless
Yasutaka Furukawa

AN ALGORITHM FOR TRIANGULATING 3D POLYGONS

by

Ming Zou

A thesis presented to the School of Engineering and Applied Science
of Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

December 2013
Saint Louis, Missouri

Contents

List of Tables . . . . . **iv**

List of Figures . . . . . **v**

Acknowledgments . . . . . **vii**

Abstract . . . . . **ix**

1 Introduction **1**
1.1 Background 1
1.2 Contributions 3
1.3 Related Work 3
1.3.1 Triangulating a single polygon 3
1.3.2 Triangulating multiple polygons 4

2 Algorithm **6**
2.1 Single polygon 6
2.2 Multiple polygons 8
2.2.1 Domains 9
2.2.2 Topologically correct triangulation 10
2.2.3 Minimal sets 14
2.2.4 Complexity analysis 15

3 Delaunay-Restricted Search **18**
3.1 Complexity 19
3.2 Existence of solutions 20

4 Experiments **22**
4.1 T: all triangles 22
4.2 T: Delaunay triangles 24
4.2.1 Optimality 29
4.2.2 Delaunay triangulability 31

5 Applications **33**

6 Conclusion **37**

6.1 Limitations 37
6.2 Future work 37

Appendix A Proof of Lemma 2.2.1 . . . . . **39**

Appendix B Proof of Lemma 2.2.2 . . . . . **41**

Appendix C Proof of Lemma 2.2.3 . . . . . **42**

References **43**

List of Tables

3.1 Worse-case time and space complexity of the algorithm in the unrestricted and Delaunay-restricted search space respectively. 19

List of Figures

1.1 Triangulations computed by our algorithm on sketched curves (left and middle) and hole boundaries with islands (right). 2

2.1 Domain splitting in a single polygon for minimizing per-triangle (a,b) and bi-triangle (c,d) weights. 7
2.2 Domain splitting in multiple polygons for optimizing per-triangle weights. . . 9
2.3 Domain splitting that results in sub-domains sharing two common vertices (v, v'). Weak edges are indicated by dashed gray lines. 11
2.4 Triangulations of two polygons (a) with a non-manifold edge (which connects the disks that fill the two polygons) (b) and with only manifold edges (c). . . 12

3.1 Complete triangle set (left) and restricted Delaunay triangle set (right) of a space curve. 19
3.2 A hexagon that is not triangulable (a), and two views of a triangulable 24-vertex polygon that is not Delaunay-triangulable (b,c). 20

4.1 Top: triangulation of a monkey saddle with minimal sum of dihedral angles. Middle and bottom: running time (left) and space usage (right) using all triangles (middle) or Delaunay triangles (bottom). 23
4.2 Top: examples of triangulations of multiple (minimizing total dihedral angles) curves used in our tests. Middle and bottom: running time (left), space usage (middle), and average number of weak edge sets in each domain (right) for these curves with increasing number of points n when using all triangles (middle) and Delaunay triangles (bottom). 24
4.3 The performance of our algorithms on polygons sampled from algebraic curves. We use area as the per-triangle metric, and dihedral angle as the bi-triangle metric. The Delaunay triangles in *Moment* and *Twist* were not generated by Tetgen, which gave numerical errors (possibly due to the large difference in the scale of coordinates) 26
4.4 Examples of randomized loops on meshes (left), and plots of performance of our algorithms (right). 27
4.5 1st and 3rd row: examples of triangulations of boundary-island polygons and parallel polygons (minimizing total dihedral angles). 2nd and 4th row: running time (left), space usage (middle), and average number of weak edge sets in each domain (right) for these polygons with increasing number of points n. 29

4.6 Histograms of area (top) and dihedral (bottom) optimality on randomly generated single loops (left) and loop pairs (right). 30
4.7 Two polygons (a) whose area-minimizing triangulation in the space of all triangles (b) has much smaller area than that in the space of Delaunay triangles (c), although the former contains intersecting triangles. 31
4.8 Hole boundaries covering various ratios of the surface area that are not Delaunay triangulable. 32

5.1 Variations of triangulations generated by our algorithm for *Spiral* (top) and *Monkey* (middle). Prescribed boundary normals, if present, are shown as arrows. 34
5.2 ILove Sketch curve data [5] with resolved patches [1] (top) is first triangulated minimizing the dihedral angle bending using our method. The surface is then refined using the method of [19], and a final boundary normal conforming bi-Laplacian surface is produced(shown bottom) using the method of Andrews et. al [3]. 35
5.3 Filling a model with numerous holes. Close-ups on a portion of the model are shown at the bottom. 36

Acknowledgments

I would like to send my utmost gratitude to my advisor Tao Ju. Without him, none of the work described in this thesis would have been possible. Tao always keeps himself available for discussing ideas and bouncing off helpful feedbacks. His kindness warms me since the first day; his enthusiasm for research and encouragement have kept me continuing my work.

I would like to thank Nathan Carr, our amazing collaborator from Adobe. Every meeting with Nathan brings us invaluable insights. Nathan has been extremely patient and kind, providing us with professional suggestions and knowledge. Working with Nathan is both enjoyable and beneficial. I would also like to thank my other committee members Robert Pless and Yasutaka Furukawa for their helpful suggestions.

I would like to thank my group members, Yixin Zhuang, Michelle Vaughn and Derek Burrows for collaborating on projects, sharing ideas, and generously taking time to help with my writings and talks.

I would like to thank my parents and grandparents for their unconditional love and support, making me feel home no matter how far away I go. I am also very thankful to my uncle who has faith in me when I have doubts, and who kept encouraging me to pursue postgraduate study. Last but not least, I thank my love Wenlin Chen for sharing with me all the joys and sorrows in the life, and supporting me through the whole time.

Ming Zou

Washington University in Saint Louis
December 2013

Dedicated to my grandpa.

ABSTRACT OF THE THESIS

AN ALGORITHM FOR TRIANGULATING 3D POLYGONS

by

Ming Zou

Master of Science in Computer Science

Washington University in St. Louis, December 2013

Research Advisor: Professor Tao Ju

In this thesis, we present an algorithm for obtaining a triangulation of multiple, non-planar 3D polygons. The output minimizes additive weights, such as the total triangle areas or the total dihedral angles between adjacent triangles. Our algorithm generalizes a classical method for optimally triangulating a single polygon. The key novelty is a mechanism for avoiding non-manifold outputs for two and more input polygons without compromising optimality. For better performance on real-world data, we also propose an approximate solution by feeding the algorithm with a reduced set of triangles. In particular, we demonstrate experimentally that the triangles in the Delaunay tetrahedralization of the polygon vertices offer a reasonable trade off between performance and optimality.

Chapter 1

Introduction

1.1 Background

In many computer graphics applications, one needs to find a surface that connects one or multiple (closed) boundary polygons. For example, such a surface is needed to fill in a hole on an incomplete mesh, and complex holes may have multiple identified boundaries (e.g., an outer boundary and several interior islands) [4]. In contour interpolation [20], 2D curves from adjacent planar sections need to be connected by one or multiple surfaces. Last but not least, with the availability of sketch-based modeling tools [5], there is an increasing demand in producing fair-looking surfaces from user-drawn curve sketches. Figure 1.1 shows some examples.

For a single boundary polygon, a common practice for surfacing is to first generate an initial mesh, usually a *triangulation* involving only the vertices on the polygon, and then to refine this initial surface to achieve better smoothness or mesh quality. The initial triangulation, given a single polygon, can be computed using a simple dynamic programming algorithm [7, 9]. A nice property of this algorithm is that it is guaranteed to produce an *optimal* triangulation that minimizes the sum of certain "weight", being either a quantity that can be measured for each individual triangle (e.g., area) or for each pair of adjacent triangles (e.g., dihedral angle). The optimality of triangulation is important, since the success of mesh refinement often depends on the quality of the initial mesh.

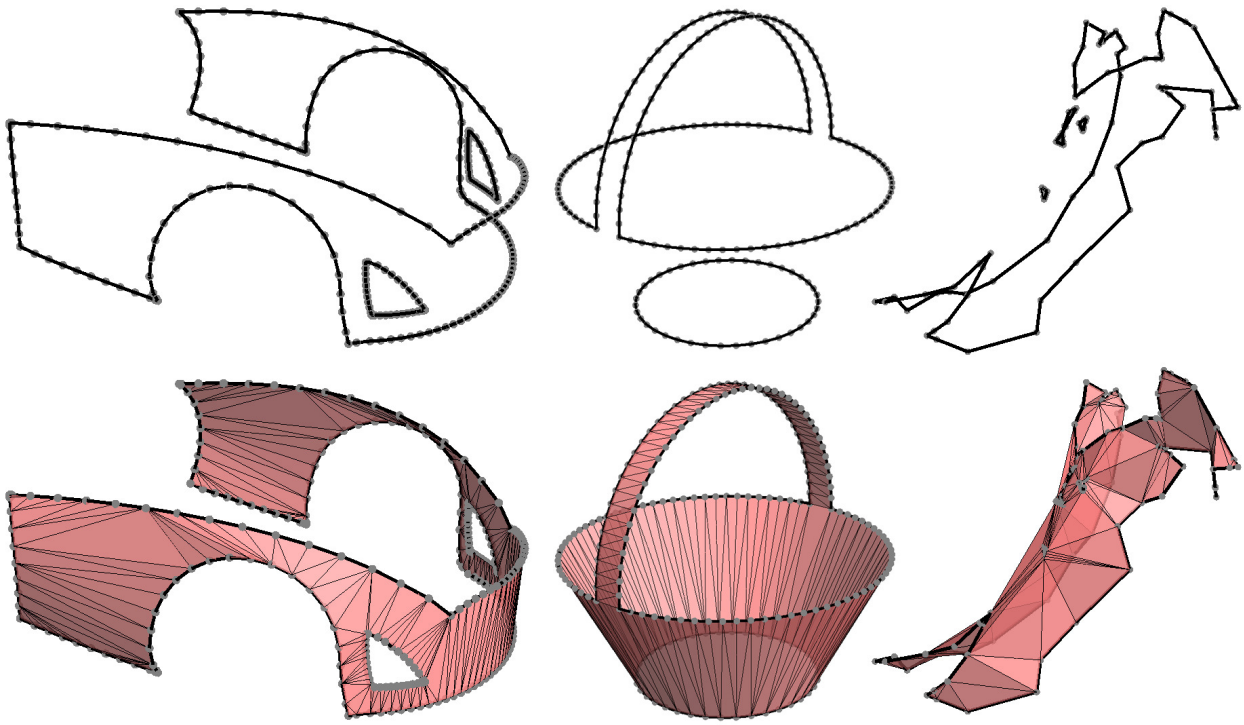

Figure 1.1: Triangulations computed by our algorithm on sketched curves (left and middle) and hole boundaries with islands (right).

In this paper, we propose an algorithm for triangulating multiple boundary polygons. The algorithm follows the divide-and-conquer strategy of [7,9], composing the optimal triangulation of a larger domain from triangulations of smaller sub-domains. A unique challenge that arises in the multi-polygon case is that simple divide-and-conquer can produce non-manifold output (i.e., edges used by more than 2 triangles). We propose a solution that avoids non-manifold edges without compromising the optimality of the result. More precisely, given k boundary polygons and a definition of weight, our algorithm finds the triangulation that minimizes the sum of weights among all triangulations whose topology is equivalent to a sphere with k holes.

A practical limitation of optimal algorithms (both [7,9] and ours) is their high computational cost. Without changing the algorithms, we explore a solution by feeding the algorithms with a *reduced* set of triangles. The choice of this reduced set has to be carefully made; the set should have significantly fewer triangles than the complete space of triangles, but still big enough to contain a near-optimal triangulation. A natural choice is the set of triangle faces

of a Delaunay tetrahedralization of the input polygons. We devised experiments to analyze the trade-off between efficiency and quality for triangulating in the Delaunay space.

1.2 Contributions

To our knowledge we are the first to develop a practical algorithm for triangulating multiple non-planar 3D space curves. We make the following two main technical contributions:

- We generalize the dynamic programming algorithm [7, 9] from a single polygon to multiple polygons with the guarantee of producing optimal, manifold triangulations. The time and space complexity of the algorithm is analyzed and validated by experiments.
- We explore the use of Delaunay triangles as a reduced input to our algorithm, and we perform experiments to demonstrate that this choice makes a good compromise between computational cost and quality of results.

1.3 Related Work

1.3.1 Triangulating a single polygon

Triangulating a simple 2D polygon is a well-studied problem in computational geometry. Many efficient algorithms have been proposed to find *a* triangulation [12, 23], although the optimality of such triangulation is not guaranteed.

Gilbert [15] and Klincsek [18] independently developed an $O(n^3)$ time and $O(n^2)$ space algorithm for computing the triangulation of a 2D polygon that minimizes the sum of total edge lengths (also known as the *minimum weight triangulation*). The algorithm, based on dynamic programming, constructs the optimal triangulation of a larger domain from the optimal triangulations of smaller sub-domains. The algorithm was extended by Barequet and Sharir [9] to find an optimal triangulation of a 3D polygon that minimizes the sum of

per-triangle weights (e.g., area), with the same complexity. A further extension [7] minimizes the sum of bi-triangle weights (e.g., dihedral angle) and uses $O(n^4)$ time and $O(n^3)$ space.

There are other approaches for triangulating a single 3D polygon but they do not possess any guarantee of optimality and are often restricted to special classes of inputs. Liepa modified the algorithm of [9] to simultaneously, but greedily minimize both the total surface area and the worst dihedral angle [19]. When the polygon is sufficiently planar, one can first project the polygon to a best-fitting plane, triangulate the planar projection, and finally lift the triangles to 3D [22]. However, the method is not applicable for curly polygons with no intersection-free planar projections. For smooth curve sketches provided by designers, the method of Rose et al. [21] extracts near-developable surfaces using convex hulls while the method of Bessmeltsev et al. [10] builds quadrangulations by interpolating the sketches with flow lines. However, it is unclear how these methods would perform on more general inputs such as jagged hole boundaries on incomplete meshes.

Note that all above methods produce outputs that may contain intersecting triangles. Determining whether a 3D polygon has a non-intersecting triangulation is in itself an NP-hard problem [7].

1.3.2 Triangulating multiple polygons

To the best of our knowledge, there is no algorithm capable of computing the optimal triangulation of multiple, general 3D polygons. However, algorithms exist for special classes of polygons.

The algorithm of Gilbert and Klincsek for a single 2D polygon can be generalized to handle a given set of interior vertices in the plane (i.e., degenerate holes each consisting of a single vertex) [16]. The algorithm relies on the 2D locations of the interior vertices to determine whether they lie inside a particular sub-domain.

Given two 2D polygons on parallel planes, the dynamic programming algorithm of Fuchs et al. [14] computes an optimal ribbon-like triangulation consisting of only triangles that span both polygons. Although the algorithm can be applied to triangulate two non-planar polygons, it does not explore those triangles whose vertices belong solely to one of the

polygons, yet such triangles are often important to produce a fair surface for non-planar inputs (e.g, the car bumper in Figure 1.1). Also, the algorithm does not generalize to multiple polygons. While many other methods can interpolate multiple planar polygons [8, 11, 20], they all rely upon the planarity of the polygons.

Chapter 2

Algorithm

We start by a brief review of the classical algorithm [7,9] for optimally triangulating a single 3D polygon. We then describe our extension to multiple polygons.

2.1 Single polygon

We consider a closed 3D polygon P and a pool of "candidate" triangles T connecting vertices of P. T can be the set of all triples of vertices of P, or some subset (see more discussions in the next section). The goal is to find a subset of T that forms a *topological triangulation* (which will be shortened as *triangulation* thereafter), which is a manifold, disk-like, but possibly self-intersecting surface with P as its boundary. Furthermore, among all triangulations, we seek an *optimal* one that minimizes the sum of some user-defined weights on each triangle (e.g., area) or between two adjacent triangles (e.g., dihedral angle).

The key idea of the algorithm is divide-and-conquer: the optimal triangulation of P is found by merging optimal triangulations of two segments of P, which are in turn constructed from optimal triangulations of smaller segments. Note that this top-down scheme is slightly different from the original bottom-up dynamic programming scheme in [7, 9]. While both schemes have the same complexity, the top-down scheme is better suited for generalization to more complex settings [16].

For the ease of explanation, let us first consider optimizing the sum of per-triangle weights (e.g., area). We define a *domain* D as a segment of P, whose end points share an edge in

the input triangle set T. We call this edge the *access edge* of D, denoted by e_D (see Fig. 2.1 (a)). Now consider a triangle t incident to e_D, whose third vertex lands on the polygon segment of D. The domain D is thus split by t into two sub-domains, D_1 and D_2, whose access edges are the two other edges of t (Figure 2.1 (b)). We can relate the weight of the optimal triangulation of a domain D, denoted as $W(D)$, to those of D_1, D_2 as:

$$W(D) = \min_{t \in T_D}(w(t) + W(D_1) + W(D_2)) \tag{2.1}$$

where T_D denotes the set of input triangles incident to e_D and whose third vertex lies on D, and $w(\cdot)$ is the user-defined weight of a triangle. As the base case, $W(D) = 0$ if D consists of only two vertices (called an *empty* domain).

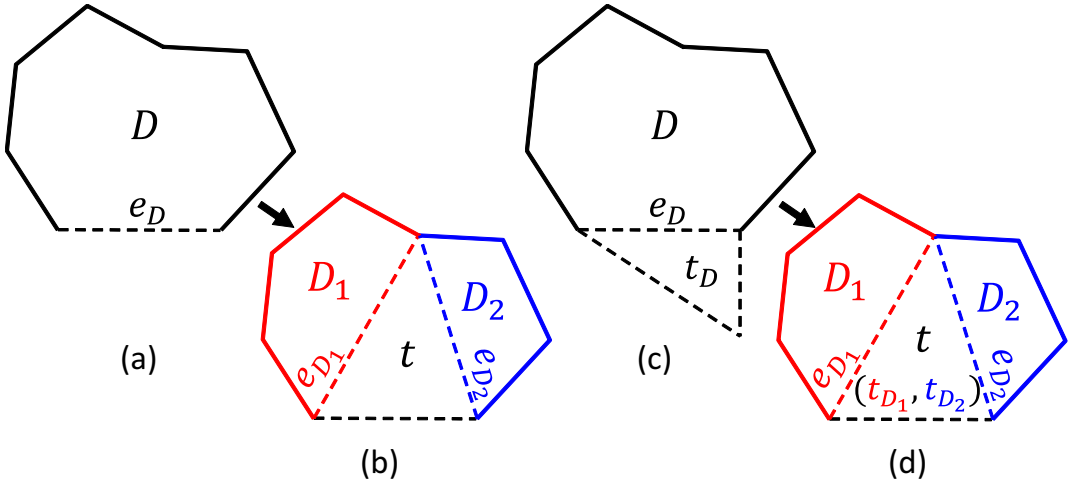

Figure 2.1: Domain splitting in a single polygon for minimizing per-triangle (a,b) and bi-triangle (c,d) weights.

If the optimization goal includes bi-triangle weights (e.g., dihedral angle), the optimal triangulation within a domain D also depends on triangles outside D. To this end, we augment our domain definition so that it is represented by both a segment of the polygon P and an *access triangle*, denoted by t_D, which is any triangle in T incident to the access edge e_D and whose third vertex lies outside D (Figure 2.1 (c)). As before, D can be split into two sub-domains D_1, D_2 by any triangle t inside D and incident to e, where t serves as the access triangle for both sub-domains (Figure 2.1 (d)). A similar relation holds between the weight

of the optimal triangulation of D, in the context of t_D, and those of sub-domains D_1, D_2:

$$W(D) = \min_{t \in T_D}(w(t) + w(t, t_D) + W(D_1) + W(D_2)) \tag{2.2}$$

where $w(\cdot, \cdot)$ (with a slight abuse of notation) is the user-defined weight between two triangles.

The relations in Equations 2.1 and 2.2 naturally lead to a recursive implementation. To avoid redundant computations, the minimal weight $W(D)$, as well as the triangle t that leads to the optimal splitting, are computed only once for each domain D and stored for subsequent look-up (a technique known as "memoization"). The recursion starts from an initial domain D that encompasses the entire polygon P except for one arbitrarily chosen access edge e_D (with an empty access triangle t_D). After the completion of the recursion, the optimal triangulation can be recovered from the splitting triangles stored at the domains.

The result of the algorithm is guaranteed to be a manifold surface with a disk-like topology. The key to this guarantee is the fact the polygon segments of D_1 and D_2 share only one common vertex. This implies that the triangulations of D_1, D_2 cannot share common edges or triangles, and neither can contain the splitting triangle t. Hence if the two triangulations are both manifold and disk-like, so is their union with t (which is a triangulation of D).

2.2 Multiple polygons

We now consider a set of polygons P_i for $i = 1, \ldots, k$. While we looked for a manifold, disk-like surface in the case of $k = 1$, here we ask the triangulation to be manifold and topologically equivalent to a sphere with k holes.

To compute the optimal triangulation, we follow the same divide-and-conquer strategy that we used before. While it is not too difficult to generalize the definition of domains to $k \geq 1$, a greater challenge is making sure that merging triangulations in these generalized domains does not introduce non-manifold edges.

We start by introducing our generalization of domains. We then present our solution to ensure topological correctness when merging triangulations, followed by a more efficient

computational approach. We end with analysis of the complexity bounds. For the ease of explanation, except for complexity analysis, we will use the sum of per-triangle weights (e.g., area) as the optimization function. Bi-triangle weights can be easily incorporated by augmenting the domains with access triangles and including the bi-triangle weight in the recursive formula (see Section 3.1).

2.2.1 Domains

A generalized domain is defined by its *boundary* and *holes* (see Figure 2.2 top-left). The boundary is a closed loop made up of one or more segments from input polygons (called *input segments*, solid edges in Figure 2.2), connected at their ends by edges in T (called *spanning edges*, dashed edges in Figure 2.2). To be able to bound the number of domains, we ask that no two input segments on the boundary of a domain come from the same polygon (so the boundary can have at most k input segments). Note that each input segment can be as small as a single vertex or as big as the entire curve. A domain may also contain input polygons (that do not appear on the boundary) as its holes. A domain is *empty* if it has no holes and if the boundary consists of only two polygon vertices, either on the same polygon or on different polygons.

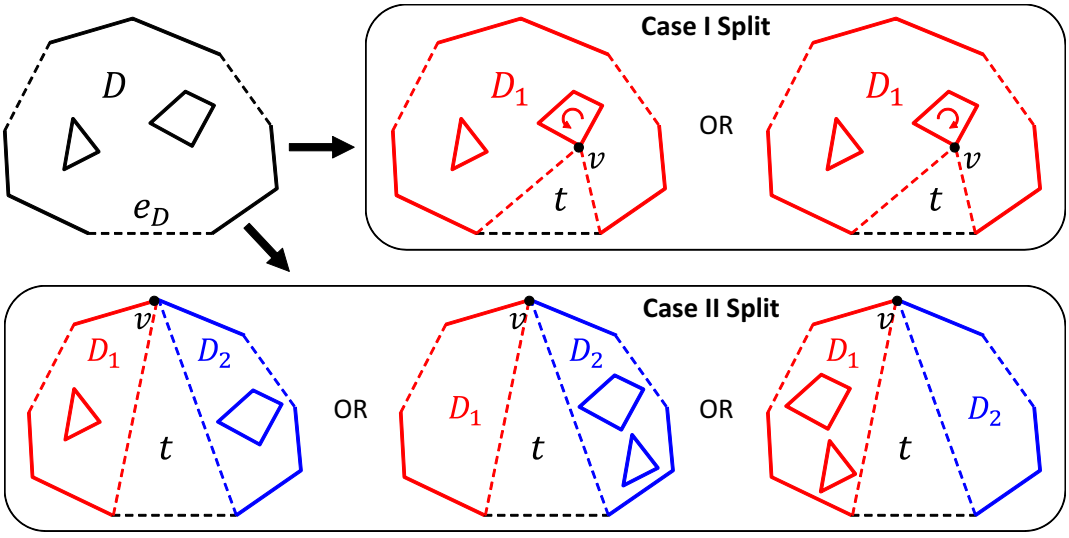

Figure 2.2: Domain splitting in multiple polygons for optimizing per-triangle weights.

To split a domain D into smaller domains, we arbitrarily assign one of the spanning edges on D's boundary as the access edge, noted as e_D, and examine all triangles t incident to e_D whose third vertex v lands on D's boundary or holes. Different from the single polygon scenario, splitting with one triangle may yield multiple combinations of sub-domains. We will separately examine two cases:

Case I: (v is on one of D's holes, P_j) Splitting results in a single sub-domain D_1 which keeps all the remaining holes of D and adds the entire P_j as an input segment to the boundary. Since there are two ways to order the edges of P_j, there are two possible D_1 (Figure 2.2 top-right).

Case II: (v is on D's boundary) Splitting results in two sub-domains D_1, D_2, each occupying a portion of D's boundary and keeping a subset of D's holes. If D has holes, there are multiple ways of distributing the holes to the two sub-domains, resulting in multiple possible pairs of D_1, D_2 (Figure 2.2 bottom).

Note that splitting always produces well-defined sub-domains where no two boundary segments come from the same input polygon. Also, a sub-domain is always "smaller" than the original domain: it either contains fewer holes (Case I) or fewer boundary vertices with no additional holes (Case II). Hence repeated splitting is guaranteed to terminate with empty sub-domains.

2.2.2 Topologically correct triangulation

With divide-and-conquer, we merge optimal triangulations of sub-domains to form the optimal triangulation of the original domain. We need to make sure that the merging does not introduce bad topology, such as non-manifold edges (used by more than 2 triangles) or tunnels.

When the input is a single polygon, we get topological correctness for free. As we explained earlier, this is because the sub-domains D_1, D_2 are disjoint except at one vertex. However, in the multiple polygon case, D_1, D_2 may share more than one common vertex, which may lead to possibly non-manifold edges after merging. To see why, observe that after a Case I split,

the third vertex v of the splitting triangle will appear twice on the boundary of the resulting sub-domain. Figure 2.3 left shows one such sub-domain (which we will refer to as D). A further Case II split of D may result in two smaller sub-domains (referred to as D_1, D_2) that both contain a copy of v, in addition to the common vertex v' of the splitting triangle in this second split (see Figure 2.3 right). If the edge $\{v, v'\}$ appears in the optimal triangulation of both D_1 and D_2, this edge would become non-manifold in the union of these triangulations. Figure 2.4 (b) shows an example of a triangulation containing a non-manifold edge.

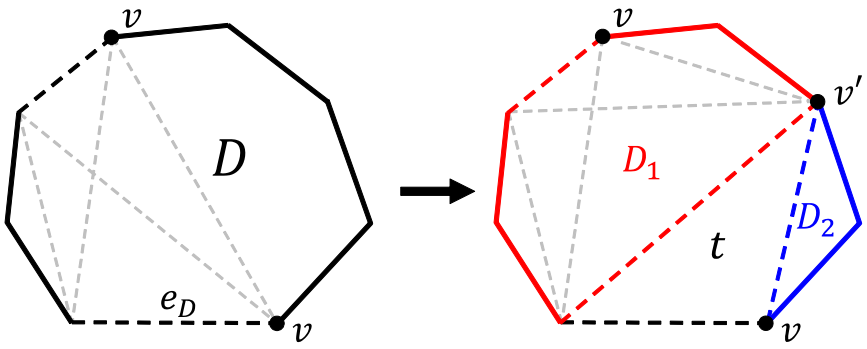

Figure 2.3: Domain splitting that results in sub-domains sharing two common vertices (v, v'). Weak edges are indicated by dashed gray lines.

A naive way to prevent such non-manifold edges is to keep track of those edges used by the optimal triangulation of one sub-domain (e.g., D_1) while computing the triangulation of the other sub-domain (e.g., D_2). However, this would break the fundamental assumption of the algorithm (that one domain can be solved independently from others).

Our key observation is that we can tell, just from the definition of a domain D, what edges in the triangulation of D may become non-manifold after merging. This is because the additional common vertices between sub-domains (e.g., v in Figure 2.3) are generated only by Case I splits, and these vertices only lie at the ends of input segments on the domain boundary. So the only edges in a triangulation of D that can possibly become non-manifold are those that connect the ends of input segments of D. These edges include all the spanning edges, which connect ends of successive input segments, as well as others that we call *weak edges* (marked in gray in Figure 2.3). For a domain with $m \leq k$ input segments on the boundary, there are at most $2m(m-1)$ weak edges. Note that a domain with a single input segment has no weak edges.

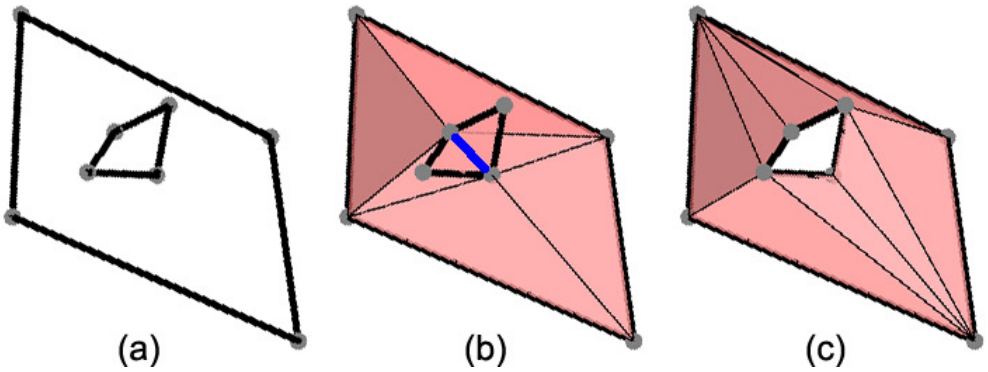

Figure 2.4: Triangulations of two polygons (a) with a non-manifold edge (which connects the disks that fill the two polygons) (b) and with only manifold edges (c).

Our strategy is to compute one optimal triangulation that uses *each combination of weak edges* in a domain. This is done in turn by optimally triangulating the sub-domains (per combination of their weak edges) and merging triangulations that don't use common weak edges. Let $E_{weak}(D)$ and $E_{span}(D)$ denote the set of all weak edges and spanning edges of a domain D, and D_1, D_2 be the sub-domains as the result of splitting with a triangle t on the access edge e_D (D_2 is empty for a Case I split).

Definition 2.2.1 *Two sets of weak edges $E_1 \subseteq E_{weak}(D_1)$ and $E_2 \subseteq E_{weak}(D_2)$ are said to be **disjoint** if sets $E_1 \cup E_{span}(D_1)$ and $E_2 \cup E_{span}(D_2)$ have no common elements and neither contains e_D. Furthermore, weak edges $E \subseteq E_{weak}(D)$ are said to be the **joint** set of E_1, E_2 if*

$$E = E_{weak}(D) \cap (E_1 \cup E_{span}(D_1) \cup E_2 \cup E_{span}(D_2) \cup \{e_D\}).$$

To compute the optimal triangulation that contains *only those* weak edges in E (and no other weak edges in $E_{weak}(D)$), we consider all optimal triangulations of sub-domains D_1, D_2 that contain disjoint weak edges whose joint set is E. That is,

$$W(D, E) = \min_{t \in T_D} \left(w(t) + \min_{\substack{\{D_1, D_2\} \in \Phi(D,t) \\ \{E_1, E_2\} \in \Psi(D_1, D_2, E)}} (W(D_1, E_1) + W(D_2, E_2)) \right) \tag{2.3}$$

Here, $W(D, E)$ is the weight of the optimal triangulation in D containing only weak edges E, $\Phi(D, t)$ is the set of all possible combinations of sub-domains as the result of splitting by triangle t, and $\Psi(D_1, D_2, E)$ is the set of all pairs of disjoint weak edge sets in D_1, D_2 whose joint set is E.

Next we show that $W(D, E)$, as defined above, is indeed minimal among all topologically valid triangulations of D containing only weak edges E. We first define what it means for a triangulation to be "valid":

Definition 2.2.2 *A triangulation S of a domain D is said to be **valid** if it meets the following two criteria:*

1. *(Manifoldness) All edges on D's boundary and holes have valence 1 in S, while the remaining edges in S have valence 2.*
2. *(Simple topology) Let h be the number of holes of D. S is homeomorphic to a sphere with $h + 1$ holes, where the boundary curve of one of the holes is homeomorphic to the boundary of D.*

Note that the second criteria is slightly different from what we desire for the complete triangulation of the input, since the boundary of a domain may have a non-trivial topology itself. For the initial domain (whose boundary is all edges on one of the polygons except for the access edge and whose holes are the remaining polygons), the second criteria is equivalent to having the topology of a sphere with k holes.

Our argument is based on the following two claims (which we prove in the Appendix). The first states that merging valid triangulations of sub-domains always results in a valid triangulation of the larger domain. The second states the inverse, which is that a valid triangulation of the larger domain can always be decomposed into valid triangulations of the sub-domains. Putting this all together, and by induction, they support that $W(D, E)$ as defined in Equation 2.3 is minimal in the space of all valid triangulations.

Lemma 2.2.1 *Let D_1, D_2 be sub-domains of D after splitting with triangle t. Let S_1, S_2 be some valid triangulations of D_1, D_2, respectively, that contain only weak edges E_1, E_2, and*

suppose E_1, E_2 are disjoint and have joint set E. Then the union $S_1 \cup S_2 \cup \{t\}$ is a valid triangulation of D containing only weak edges E.

Lemma 2.2.2 *Let S be a valid triangulation of D containing only weak edges E, and t be the triangle in S incident to the access edge e_D. Then there exist some sub-domains $\{D_1, D_2\} \in \Phi(D, t)$ and sets $\{E_1, E_2\} \in \Psi(D_1, D_2, E)$, such that $S \setminus \{t\}$ is made up of two valid triangulations in D_1, D_2, respectively, containing only weak edges E_1, E_2.*

2.2.3 Minimal sets

A practical problem with implementing Equation 2.3 in a recursive program is the potentially high computational cost. Even though the number of weak edges is small within a domain, the number of possible combinations of them (E) can be prohibitively large. Since a domain may contain up to $2k(k-1)$ weak edges, it may have up to $2^{2k(k-1)}$ subsets. For $k = 3$, this number is already 4096. Although some of these subsets do not give rise to any triangulation (e.g., those that involve "crossing" weak edges in Figure 2.3), most of these subsets do. The number of weak edge combinations affects both the time and space usage of the program, as the weight $W(D, E)$ needs to be computed and stored in memory for each combination E.

The following observation helps reducing the number of weak edge combinations that need to be explored. Consider two triangulations S, S' of a same domain D that use weak edges E, E' respectively. If the weight of S is less than that of S', *and* if E is a subset of E', then *any* triangulation of the input polygons that uses S' can instead use S in place of S' to have a lower total weight without violating the topological validity. This is because S has less chance of causing "conflicts" with the rest of the triangulation than S'.

With this observation, the program only needs to explore (and store) a weak edge set if none of its subsets gives rise to a lower-weight triangulation. We call such set *minimal weak edge set*, or simply *minimal set*. We further denote the set of all minimal sets for a domain D, under a certain weight definition W, as $\Pi(D, W)$. Formally,

$$\Pi(D, W) = \{E \subseteq E_{weak}(D) | \forall E' \subset E, W(D, E') > W(D, E)\}$$

Now we can modify Equation 2.3 to restrict the enumeration of weak edges sets to only minimal sets,

$$\hat{W}(D,E) = \min_{t \in T_D} \left(w(t) + \min_{\substack{\{D_1,D_2\} \in \Phi(D,t) \\ \{E_1,E_2\} \in \Psi(D_1,D_2,E) \\ E_1 \in \Pi(D_1,\hat{W}) \\ E_2 \in \Pi(D_2,\hat{W})}} (\hat{W}(D_1,E_1) + \hat{W}(D_2,E_2)) \right). \tag{2.4}$$

We need to show that the restriction to minimal sets in Equation 2.4 does not compromise optimality. This is stated in the next lemma (see proof in Appendix):

Lemma 2.2.3 *Let W and W' be defined recursively using Equations 2.3 and 2.4, respectively. Then, for any domain D,*

1. $\Pi(D,W) = \Pi(D,\hat{W})$
2. *For all $E \in \Pi(D,W)$, $W(D,E) = \hat{W}(D,E)$.*

The pseudo-code for the recursive program that computes minimal weights $\hat{W}$ and minimal sets Π within a domain is given in Algorithm 1.

2.2.4 Complexity analysis

We give an upper bound of time and space used by the algorithm. Let n be the total number of vertices in the input polygons, n_E, n_F be the total number of edges and triangles in the input triangle set T.

We first bound the number of domains, denoted as n_D. When minimizing per-triangle weights, each domain contains at most k spanning edges that appear in T. So n_D is upper bounded by $O(n_E^k)$. To minimize bi-triangle weights, each domain needs to be augmented by adding a triangle in T for each spanning edge, which brings the bound to $O(n_F^k)$.

Algorithm 1 procesDomain(D)

Input: Domain D
Output: $\Pi[D]$ // minimal weak edge sets of domain D
Output: $W[D, E]$ // minimal weight of triangulating D using weak edges E

if $\Pi[D]$ already exists **then**
    return
end if
$\hat{W}[] \leftarrow \infty$ //temporary weights for each weak edge set
$S \leftarrow \emptyset$ //temporary list of weak edge sets
for each $t \in T(D)$ incident to e_D **do**
    for each $\{D_1, D_2\} \in \Phi(D, t)$ **do**
        processDomain(D_1)
        processDomain(D_2)
        for each disjoint pair $E_1 \in \Pi[D_1], E_2 \in \Pi[D_2]$ **do**
            $E \leftarrow$ joint set of E_1, E_2
            $\hat{W}[E] \leftarrow Min(\hat{W}[E], w(t) + W[D_1, E_1] + W[D_2, E_2])$
            $S \leftarrow S \cup \{E\}$
        end for
    end for
end for
$\Pi[D] \leftarrow \infty$
for each $E \in S$ **do**
    if there is no $E' \in S$ s.t. $E' \subset E$ and $\hat{W}[E'] < \hat{W}[E]$ **then**
        $\Pi[D] \leftarrow \Pi[D] \cup \{E\}$
        $W[D, E] \leftarrow \hat{W}[E]$
    end if
end for

Each recursive call to processDomain() in the pseudo-code of Algorithm 1 is dominated in time by the three nested "for" loops, whose complexity have bounds $O(n)$ (choosing t) ,$O(2^k)$ (choosing $\{D_1, D_2\}$), and $2^{O(k^2)}$ (choosing pair $\{E_1, E_2\}$). Hence the time complexity is bounded by $O(n_D n 2^{O(k^2)})$. The space usage is dominated by the storage for minimal weights for each minimal weak edge set at each domain, and hence it is bounded by $O(n_D 2^{O(k^2)})$.

If we treat k as a constant, the time and space bounds become $O(n_D n)$ and $O(n_D)$ respectively. When using all possible triangles as the input set T, the {time,space} bounds are $\{O(n^{2k+1}), O(n^{2k}\}$ when minimizing per-triangle weights, and $\{O(n^{3k+1}), O(n^{3k})\}$ when minimizing bi-triangle weights. Note that these bounds agree with those of the classical dynamic programming algorithms for $k = 1$.

Chapter 3

Delaunay-Restricted Search

Our optimal algorithm guarantees that the result surface has the minimum sum of metrics over all the surface triangles. However, an important practical limitation of the optimal algorithm is its high computational cost, particularly for multiple curves. In our implementation, we found that the algorithm would take minutes and gigabytes of memories for just a few hundred vertices.

One way to avoid this prohibitive cost of optimal triangulation, is to feed the algorithm with fewer triangles T. The choice of this reduced set has to be carefully made. It should be considerably smaller than the space of all possible triangles, so that we get a notable boost in performance. On the other hand, the set should still be big enough to contain some close-to-optimal triangulation.

We consider the set of triangle facets in the Delaunay tetrahedralization of the polygon vertices (which we abbreviate as *Delaunay triangles*), for three reasons. First, there are much fewer triangles in Delaunay search space. For n points, the number of Delaunay triangles is bounded by $O(n^2)$, comparing to $O(n^3)$ in un-reduced space, as shown in Figure 3.1. Second, since Delaunay triangles tend to connect nearby vertices, T is likely to contain low-weight triangulations if the optimal triangulation also connects nearby parts of the polygons. Finally, as an added benefit, any triangulation computed in T is free of self-intersections, since no two Delaunay triangles are intersecting.

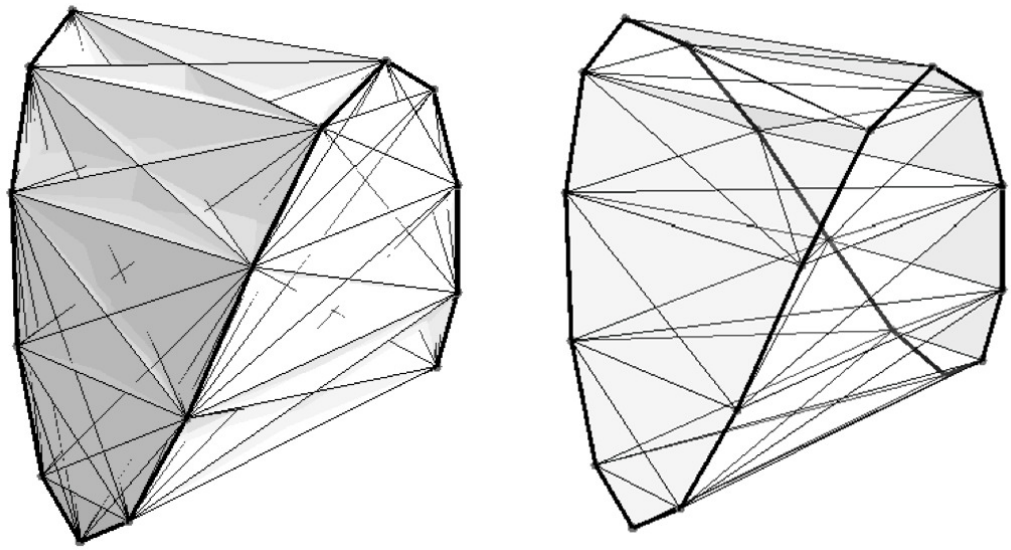

Figure 3.1: Complete triangle set (left) and restricted Delaunay triangle set (right) of a space curve.

3.1 Complexity

There are only $O(n^2)$ Delaunay triangles for n points in $\mathbb{R}^3$, in contrast to $O(n^3)$ triangles in the un-reduced space. This reduces the worst-case bounds on time and space by a factor of n, as shown in Table 3.1 (assuming k is a constant). The only exception is, when $k = 1$, the space complexity for minimizing per-triangle metrics is still $O(n^2)$ on Delaunay-restricted triangle set.

Since the Delaunay tetrahedralization can be computed in expected $O(n^2)$ time [13], the end-to-end running time of triangulation using Delaunay triangles are still bounded by $O(n^{2k+1})$ and $O(n^{3k+1})$ respectively for minimizing per-triangle and bi-triangle metrics.

	Per-triangle metrics (unres./Delaunay-res.)	Bi-triangle metrics (unres./Delaunay-res.)
Time	$O(n^{2k+1})$ / $O(n^{k+1})$	$O(n^{3k+1})$ / $O(n^{2k+1})$
Space	$O(n^{2k})$ / $O(n^{k})$	$O(n^{3k})$ / $O(n^{2k})$

Table 3.1: Worse-case time and space complexity of the algorithm in the unrestricted and Delaunay-restricted search space respectively.

3.2 Existence of solutions

Since not all 3D polygons are triangulable, a polygon may not have any triangulation that is restricted to the Delaunay triangles. We say that such polygons are not *Delaunay-triangulable*. For these polygons our algorithm will fail to return any solution. Figure 3.2 show two such examples. The hexagon in (a) was found by Barequet et. al. [7] and is a 3D polygon with the fewest vertices that cannot be triangulated (the authors proved that any polygon with vertices fewer than 6 is triangulable). The polygon in (b) is a spiral square coil where the square turns have non-equal diameters. The polygon is triangulable, according to the sufficient conditions proven in [7], because it has a non-intersecting projection the view plane in (c). However, our algorithms return no solution for this input.

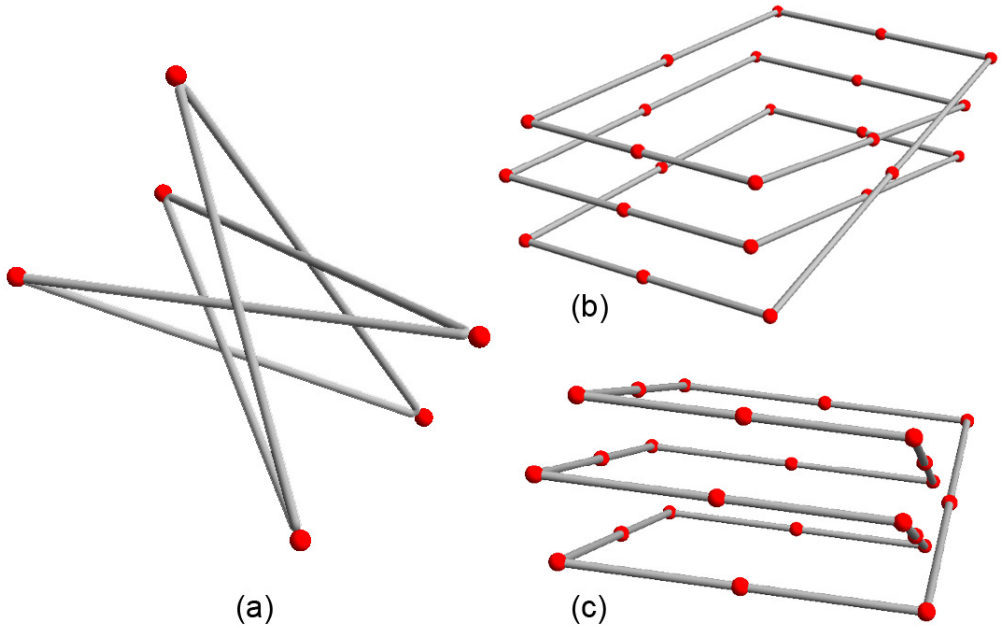

Figure 3.2: A hexagon that is not triangulable (a), and two views of a triangulable 24-vertex polygon that is not Delaunay-triangulable (b,c).

To evaluate whether our algorithms are suited for a particular application, it would be useful to have some geometrically characterizations of the kind of inputs that are Delaunay-triangulable. We next offer some preliminary results in this direction by giving a few necessary and sufficient conditions. Unfortunately, just like the general triangulability problem [7], we do not have conditions that are both necessary and sufficient. In the next section, we

will resort to extensive experimental results to demonstrate that most practical polygons are indeed Delaunay-triangulable.

Lemma 3.2.1 *A polygon is not Delaunay-triangulable if*

1. *it is knotted, or*
2. *some polygon edges are not Delaunay (i.e., not incident to any Delaunay triangles).*

Proof: It is obvious that each edge of the input polygon needs to be incident to some Delaunay triangles for a triangulation to exist. Also, it was shown that any knotted polygon is not triangulatable [7], which also implies that it is not Delaunay-triangulable. □

It is worth pointing out that the two necessary conditions are fairly mild in practice. If the input polygon is intended to bound some disk-like surface, then the polygon cannot be knotted. If it is the second case, that edges in the input polygons are not always Delaunay, meaning that some edges may not be incident to any Delaunay triangles, there are also techniques to turn those edges to be all Delaunay. A remedy is introducing additional vertices to subdivide such edges, so that the subdivided segments are Delaunay. Shewchuk gave a provably good algorithm for doing so [24] (called *edge protection*), which is guaranteed to terminate without producing excessively short segments. Actually, any 3D polygon can be turned into one whose edges are all Delaunay by subdividing the polygon edges. Alternatively, if the polygon is sampled from some smooth space curve (which is typical for sketch inputs), the polygon edges are always Delaunay as long as the samples are placed denser than the local feature size of the curve (Theorem 12 in [2]–although the theorem was stated in 2D, the statement and the proof can be easily extended to higher dimensions).

We found that edges in all our test inputs made up of polygons sampled from smooth curves are already Delaunay. Only a small fraction (less than 1%) of randomly generated polygons (see discussion next) require edge protection. For these inputs, we see an increase of number of vertices by at most a factor of 2 after edge protection. While subdividing input edges is not ideal, this usually would not cause a significant issue for downstream applications. For example, if the triangulation serves as a hole filler on a mesh, one only needs to subdivide the mesh triangles incident to the protected edges to maintain a watertight surface.

Chapter 4

Experiments

The algorithm is implemented in C++. For generality, the implementation optimizes for both per-triangle and bi-triangle weights (hence the complexity bounds follow those of the bi-triangle weights). We experimented with different choices of the input triangle set T. All experiments were done on a 6-core 3.0GHz workstation with 12GB memory.

4.1 T: all triangles

We first let T be all triples of vertices in the input polygons. In the first test, we consider a single polygon whose vertices are uniformly sampled on a "monkey saddle" curve (Figure 4.1 top). Since the time and space usage of the optimal algorithm depend only on the number of vertices in the polygon and not on the actual geometry, it suffices to test on one data set for a fixed number of polygons. The running time and space usage of the algorithm are plotted in Figure 4.1 middle. The exponents in the best-fitting polynomial of n are 4.11 and 2.27 respectively for the time and space graphs, which agree with their theoretical bounds ($O(n^4)$ and $O(n^3)$). As projected by the plots, it would take gigabytes of memory and tens of minutes to triangulate several hundred points.

In the second test, we consider multiple (up to six) polygons generated as follows. Each polygon has the same number of vertices which are sampled uniformly from a regular saddle curve. We first place the polygons at the six corners of a regular octahedron and then apply a slight random rotation and translation to each polygon. We consider a subset of $k = 2$ to 6 of these polygons as the input. The inputs for various k are shown in Figure 4.2 top.

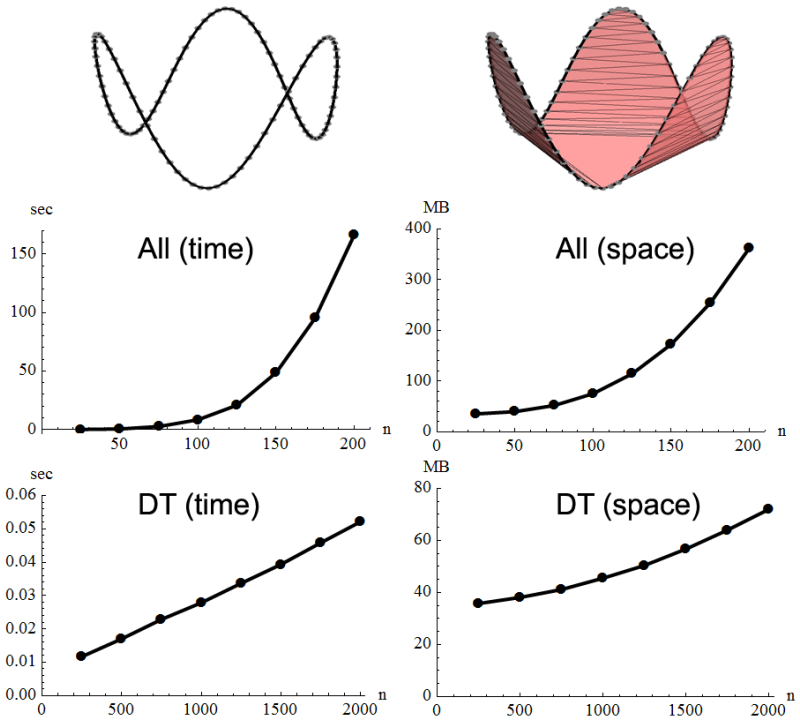

Figure 4.1: Top: triangulation of a monkey saddle with minimal sum of dihedral angles. Middle and bottom: running time (left) and space usage (right) using all triangles (middle) or Delaunay triangles (bottom).

The scalability of the algorithm drops significantly with k (Figure 4.2 middle). Triangulating 2 polygons with a total of only 40 vertices, or 3 polygons with a total of only 20 vertices, takes roughly the same time with triangulating a single polygon with 200 vertices and consumes more than 1 gigabyte of memory (results for $k > 3$ are not shown). The estimated exponent of n in {time, space} are $\{6.97, 4.99\}$ for $k = 2$ and $\{10.53, 7.71\}$ for $k = 3$, which again agree with the theoretical worst-case bounds ($\{O(n^{3k+1}), O(n^{3k})\}$).

Note that the use of minimal sets has a significant effect on reducing the number of weak edge combinations that have to be computed and stored at a domain. As plotted in Figure 4.2 middle-right, average number of weak edge sets for each domain drops by nearly a factor of k for $k = 2, 3$ after using the minimal sets. All experiments in this paper (except for this comparison plot) were conducted using the minimal sets.

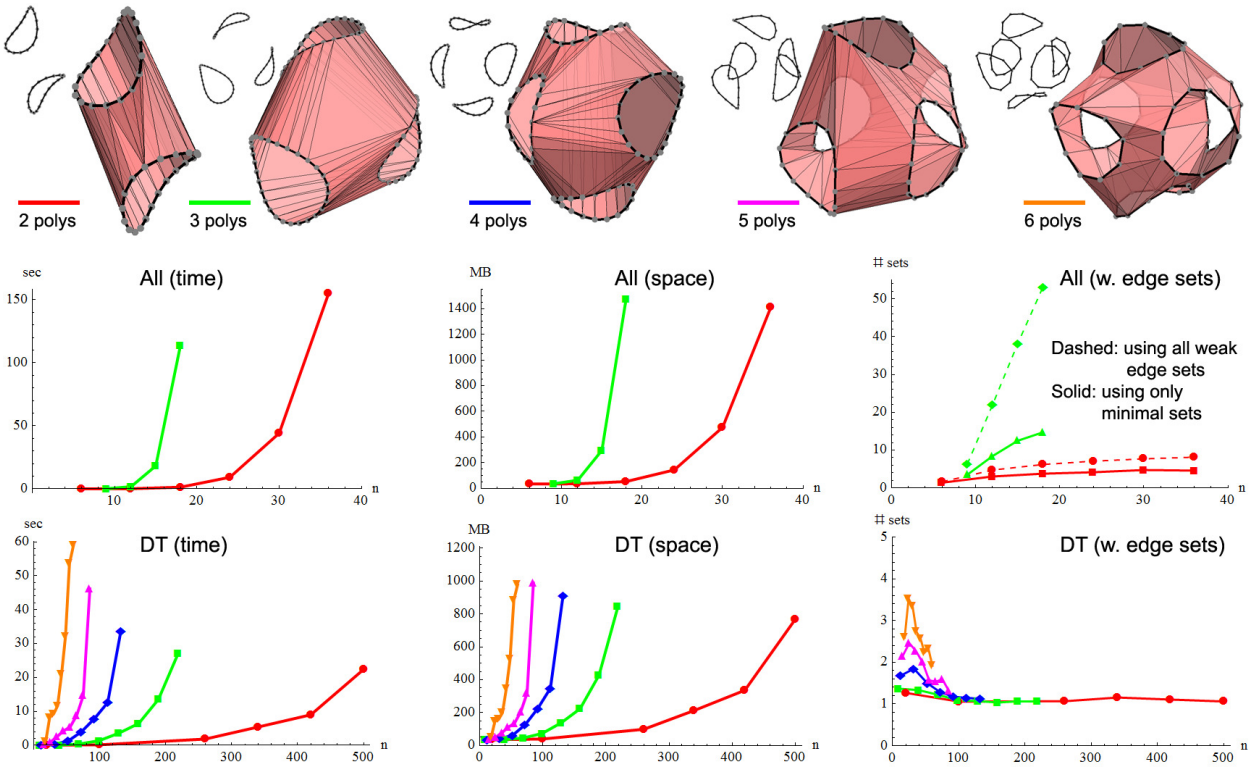

Figure 4.2: Top: examples of triangulations of multiple (minimizing total dihedral angles) curves used in our tests. Middle and bottom: running time (left), space usage (middle), and average number of weak edge sets in each domain (right) for these curves with increasing number of points n when using all triangles (middle) and Delaunay triangles (bottom).

4.2 T: Delaunay triangles

The optimal triangulation algorithm can be computational expensive in practise, particularly for multiple curves. There are many possible ways to reduce the computation. The method we adopt in this thesis is to feed the algorithm with fewer triangles T. We consider the *Delaunay triangles*: the set of triangle facets in the Delaunay tetrahedralization of the polygon vertices.

The running time and space consumption, using Delaunay triangles as T, for single and multiple polygons are plotted respectively in Figure 4.1 (bottom) and 4.2 (bottom). We use Tetgen [25] to compute the Delaunay triangles, whose running time is always less than a

second for all our inputs. For a single polygon, our algorithm can now process 2000 vertices under a fraction of a second and use less than 100 MB memory. The growth of time and space tend to be linear, in contrast to the worst-case bounds (cubic in time and quadratic in space). The performance for multiple polygons is also significantly better than before. Two curves with 250 vertices each can be triangulated in half a minute, while 6 curves with 10 vertices each can be triangulated in one minute.

Using Delaunay triangles as the input T, the time and space usage of the algorithm no longer depend only on the number of vertices in the polygon, but also depend on the actual geometry. This is because the number of Delaunay triangles varies with the shape of the polygon. To explore the impact of the polygon shape on the performance of our algorithm, we tested the algorithm on additional polygon datasets, as described bellow.

For a single polygon, we consider approximations of both smooth and piece-wise smooth algebraic curves as follows (see Figure 4.3 top):

- *Mobius*: Uniform samples on the boundary of a Mobius strip.
- *Spiral*: Uniform samples on two spirals, one shifted upwards from another, and on the straight line segments that connect the ends of the spirals.
- *Moment*: The "moment curve" whose ith vertex has coordinates $\{i, i^2, i^3\}$.
- *Twist*: A variation of the "moment curve".

We include the "moment curve" here because it was originally used to demonstrate the $O(n^2)$ upper bound of the number of simplicies in a Delaunay tetrahedralization (note that there is a tetrahedron connecting every four vertices of the form $\{i, i+1, j, j+1\}$ for any i, j). The number of samples in each example ranges from 50 to 5000.

To demonstrate the upper bound of time complexity, we prepared an additional and rather extreme example called *Twist* (Figure 4.3 top-right), which connects vertices of *Moment* in a crisscross manner. Assuming there are an odd number n of vertices on *Moment*, they are connected in *Twist* with the new order $1, n-1, 3, n-3, \ldots, 4, n-2, 2, n$. That is, take the list of vertices on *Moment* with odd, ascending indices and interleave with the list of vertices with even, descending indices.

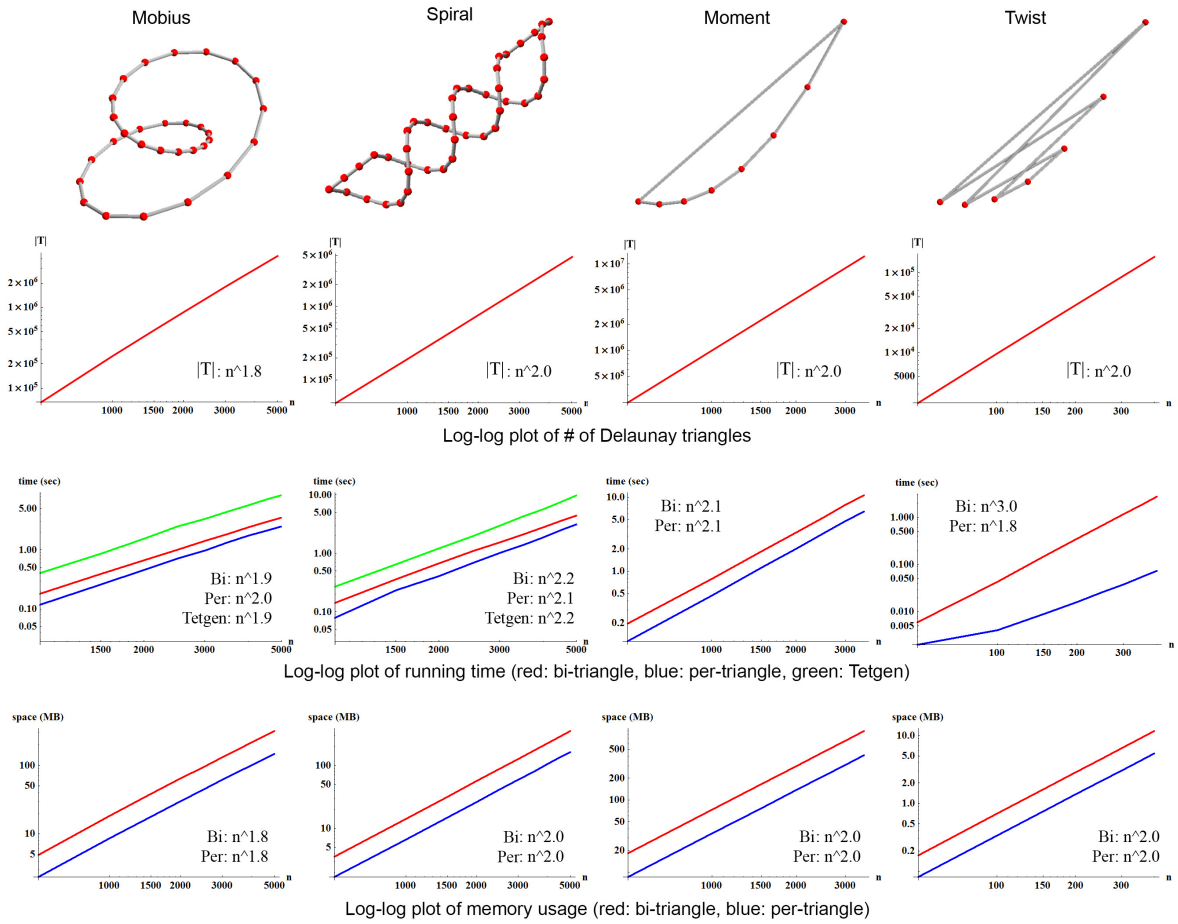

Figure 4.3: The performance of our algorithms on polygons sampled from algebraic curves. We use area as the per-triangle metric, and dihedral angle as the bi-triangle metric. The Delaunay triangles in *Moment* and *Twist* were not generated by Tetgen, which gave numerical errors (possibly due to the large difference in the scale of coordinates)

The performance of our algorithm on the polygons sampled from algebraic curves is plotted in Figure 4.3 (3rd to 4th rows). We used area metric and dihedral metric. Observe that, as the number of samples increases, the time and space, as well as the time taken by Tetgen to generate Delaunay triangles, grow at a rate no more than quadratic in all examples (except for *Twist*, which is explained below). The actual amount of time and space consumed for each example is highly correlated with the number of Delaunay triangles (2nd row).

The only outlier to the observed quadratic complexity is *Twist*, on which our algorithm uses $O(n^3)$ time, the theoretical upper bound derived in Section 3.3. To see why, recall that

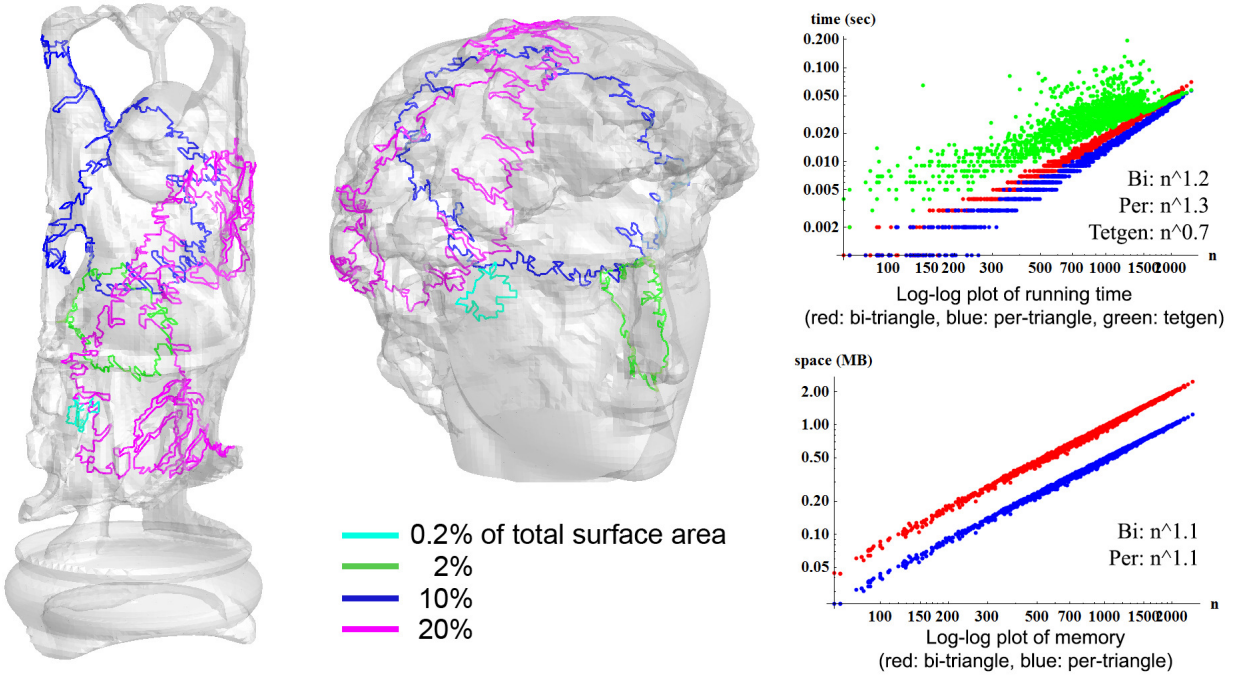

Figure 4.4: Examples of randomized loops on meshes (left), and plots of performance of our algorithms (right).

the running time of the algorithm is correlated with the number of proceeding triangles for each triangle. One can find $O(n^2)$ triangles on *Twist*, each formed by vertices $\{i, i+1, j\}$ on *Moment* for some i, j and having $O(n)$ proceeding triangles at the oriented edge $\{i, i+1\}$. Obviously, *Twist* is an extremely pathological case that rarely arises in practice. The performance data on other, more typical examples suggests the complexity of the algorithm is at most quadratic in practical inputs.

Additionally, to test the capability of algorithm on random polygons, we use the following patch-growing algorithm to produce a random set of loops with controllable size. We start with a randomly selected triangle on the mesh and grow it to a patch with a specified number of triangles. At each step of growth, we randomly select a non-patch triangle incident to the patch boundary whose inclusion in the patch does not alter the topology of the patch (in this way the patch boundary remains a single loop). The boundary loop of the final patch is used as the input to our algorithms. To produce curves with varying shapes and complexity, we used the Happy Buddha and the head portion of David from the Stanford Scanning Repository. Both meshes have non-trivial shapes and Buddha has a complex topology. For

each mesh, we generated 1000 patches each containing 100 to 10K triangles, which cover up to 20% of the surface area. Examples of loops bounding patches of various sizes are shown in Figure 4.4 (left). In comparison to typical hole boundaries in real-world data, these randomly generated loops have much more contorted shapes. However, we believe such data is useful for exploring the capability and limitation of the triangulation algorithms in restricted Delaunay search space.

Similar observations can be made for the randomized loop data set shown in Figure 4.4 (right). Each dot in the plots represents an individual loop in our data set. Observe that the growth rates of time and space are now close to being linear. Triangulation is highly efficient for this data set, taking only a fraction of a second and hardly any memory for hole boundaries containing thousands of points.

For multiple polygons, besides the test examples in Figure 4.2, we consider two more types of input that often arise in practical applications. The first type of input has a large polygon on the outside and several small island-like polygons inside the large one (Figure 4.5 1st row). This type of input is designed to mimic complex holes on incomplete meshes, which may contain small pieces of geometry inside that need to be connected to the hole boundary (an example is shown in Figure 1.1 right). We use the regular saddle curve for each of the polygons and apply a slight random rotation and translation to each polygon. We additionally stretch the boundary polygon in one direction, as holes on meshes are often elongated. To achieve a consistent sampling rate, we make the larger polygon 10 times denser than each smaller polygon. The second type of input is made up of multiple planar polygons that lie on two parallel planes (Figure 4.5 3rd row). This input is designed to mimic the scenario of interpolating contours on two cross-section planes. The triangulation can be used for reconstructing surfaces from planar contours, which is a classical problem that has been studied for decades [8, 11, 14, 20]. To test how our algorithm performs on this kind of data, we first place 3 cycles in each of the two parallel planes and add some randomness by perturbing the position of each input vertex as well as the centroid of each cycle within the plain. We then consider a subset of $k = 2$ to 6 of these polygons as the input.

The time and space usage of our algorithm, using Delaunay triangles as T, for boundary-island polygons and parallel planar polygons are plotted respectively in Figure 4.5 2nd row and 4th row. The growth of time and space appear to be significantly slower on these two

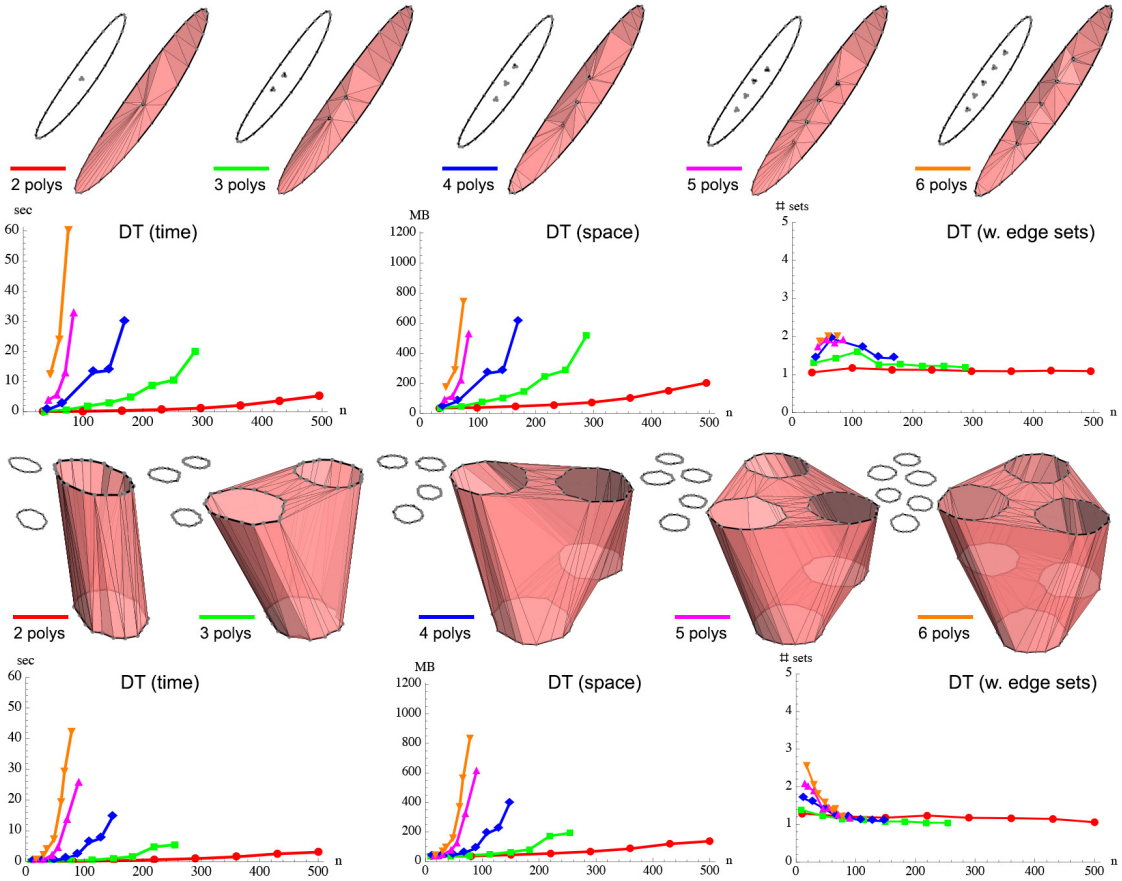

Figure 4.5: 1st and 3rd row: examples of triangulations of boundary-island polygons and parallel polygons (minimizing total dihedral angles). 2nd and 4th row: running time (left), space usage (middle), and average number of weak edge sets in each domain (right) for these polygons with increasing number of points n.

special inputs than on the uniformly distributed polygons (Figure 4.2). It also takes less absolute time for the algorithm to triangulate polygons with the same number of vertices. These results suggest that our algorithm likely perform better for practical data (e.g., mesh holes and parallel contours) than for uniformly distributed loops.

4.2.1 Optimality

A question that remains is how *good* are the triangulations computed in this reduced space. To answer this, we created a different, randomized test suite. We took a triangulated surface

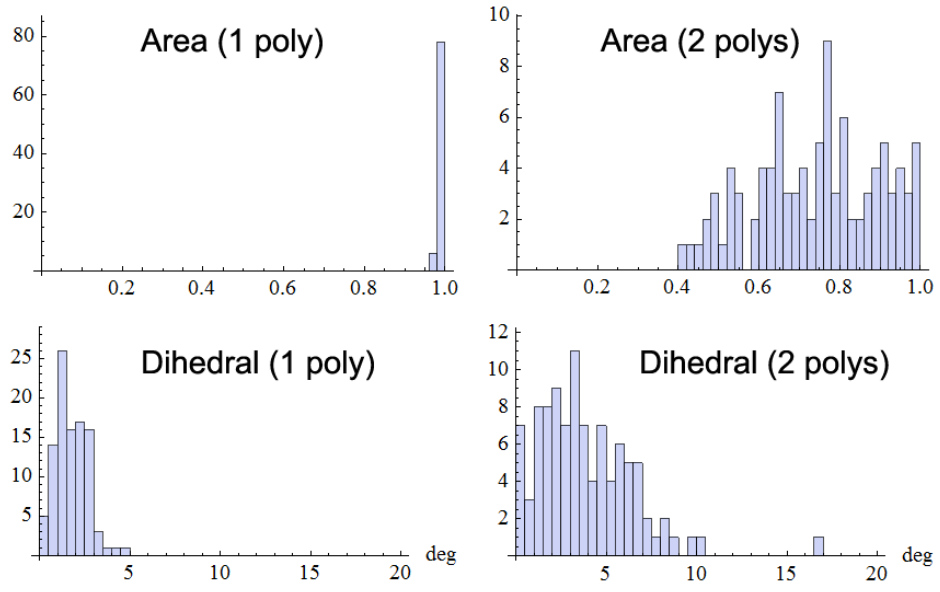

Figure 4.6: Histograms of area (top) and dihedral (bottom) optimality on randomly generated single loops (left) and loop pairs (right).

(e.g., the Bunny) and randomly generated closed, non-intersecting edge loops. We created 100 single loops each containing between 100 to 200 vertices, and 100 pairs of loops where each loop contains between 10 to 20 vertices. We then compare the weight of the triangulation computed by the algorithm using only Delaunay triangles (denoted as w_{DT}) to the weight of the triangulation computed using all triangles (denoted as w_{All}). We consider two kinds of weights, sum of triangle areas and sum of dihedral angles. For area, optimality is measured as the ratio w_{All}/w_{DT} (the closer to 1 the better). For dihedral, we measure optimality by the difference (in degrees) $(w_{DT} - w_{All})$ normalized by the number of interior edges (the closer to 0 the better).

Observe from Figure 4.6 that triangulations found using Delaunay triangles are near optimal for a single polygon using either area or dihedral weights. For two polygons, optimality of dihedral angles drops slightly (increases from 5 to 10 degrees), but optimality of total area drops sharply. To explain the latter, we note that, when using all triangles, an area-minimizing triangulation of two (or more) polygons that are far from being co-planar (such as the ones in our tests) often consist of "disks" that fill each polygon plus narrow "tunnels" connecting the disks; see an example in Figure 4.7 (b). These tunnels usually consist of self-intersecting triangles that are not Delaunay. See the area-minimizing triangulation in the

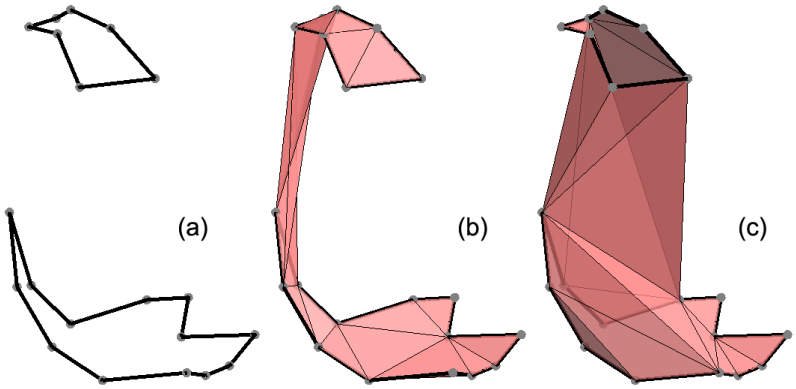

Figure 4.7: Two polygons (a) whose area-minimizing triangulation in the space of all triangles (b) has much smaller area than that in the space of Delaunay triangles (c), although the former contains intersecting triangles.

Delaunay space in Figure 4.7 (c). In contrast, dihedral-minimizing triangulations for such inputs tend to form "thicker" tunnels whose triangles are more spaced out (see examples in Figure 4.2 top) and hence more likely to be Delaunay.

In summary, our experiments show that choosing Delaunay triangles as T offers a good balance between efficiency and the optimality of results.

4.2.2 Delaunay triangulability

Our algorithms, running with Delaunay triangles, returned solutions on all of our test data except for 32 hole boundaries on the Happy Buddha. The hole in the vast majority of these failure cases (28 out of 32) covers over 10% of the surface area of the model, and the smallest hole covers 5% of the surface area. The hole boundaries all have very convoluted shapes (see Figure 4.8). Given that polygons like these typically do not arise in practice, we believe the algorithms will return solutions for the majority of the practical data, whether they are piecewise smooth curve sketches or hole boundaries on real-world meshes. However, the chance of success may decrease with increasing complexity of the polygon shape. Since Delaunay triangulability implies triangulability, our results also suggest that most practical 3D polygons are triangulable.

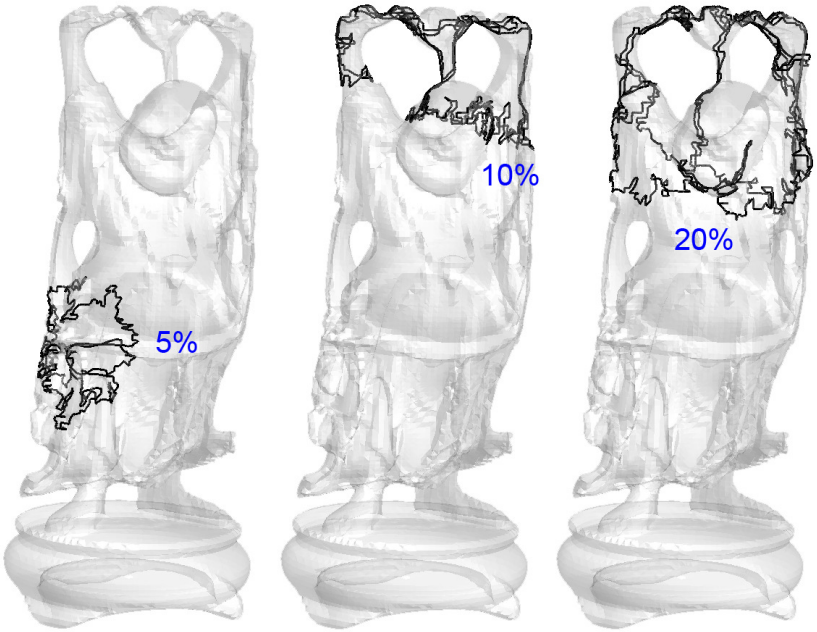

Figure 4.8: Hole boundaries covering various ratios of the surface area that are not Delaunay triangulable.

Chapter 5

Applications

The flexibility of handling different metrics and their combinations allows our algorithm to be able to generate a wide range of triangulations. In Figure 5.1 we show several variations of triangulations created for two input polygons, using objective functions such as minimal triangle areas, minimal squared dihedrals, maximal squared dihedrals (which maximize twisting), minimal deviation with boundary normals, and their combinations.

Our algorithm can be useful in a variety of applications, ranging from sketch-based modeling to hole filling on surfaces. Figure 5.2 shows the triangulation results of our algorithm on two real-world sketch inputs. Given a network of sketched spatial curves, we first apply the method of [26] to extract individual loops of curves that form the patch boundaries. Then each patch was triangulated individually by minimizing the sum of squared dihedral metric using our algorithm. Starting from the initial triangulation, we can obtain a final surface using refinement [19] and smoothing [3] operators. The results for two sketch inputs, Roadster and Spider, are shown in Figure 5.2.

While there are numerous methods available for creating surfaces bounded by a single loop in a curve sketch or filling a simple hole boundary on a mesh, creating provably good surfaces bounded by *multiple* curves is a well-known open problem, particular in hole-filling [4, 17]. Our algorithm makes a first step towards filling this gap. Figure 1.1 shows two examples of triangulating multiple sketch curves to form patches with holes. These are computed using Delaunay triangles while minimizing the sum of dihedral angles. Figure 5.3 shows an example of filling complex mesh holes. The original model has numerous holes, many of which have interior "islands" (see close-up and the right-most example in Figure 1.1). Filling such holes while interpolating the islands is a challenging task that so far defies local methods [4, 17].

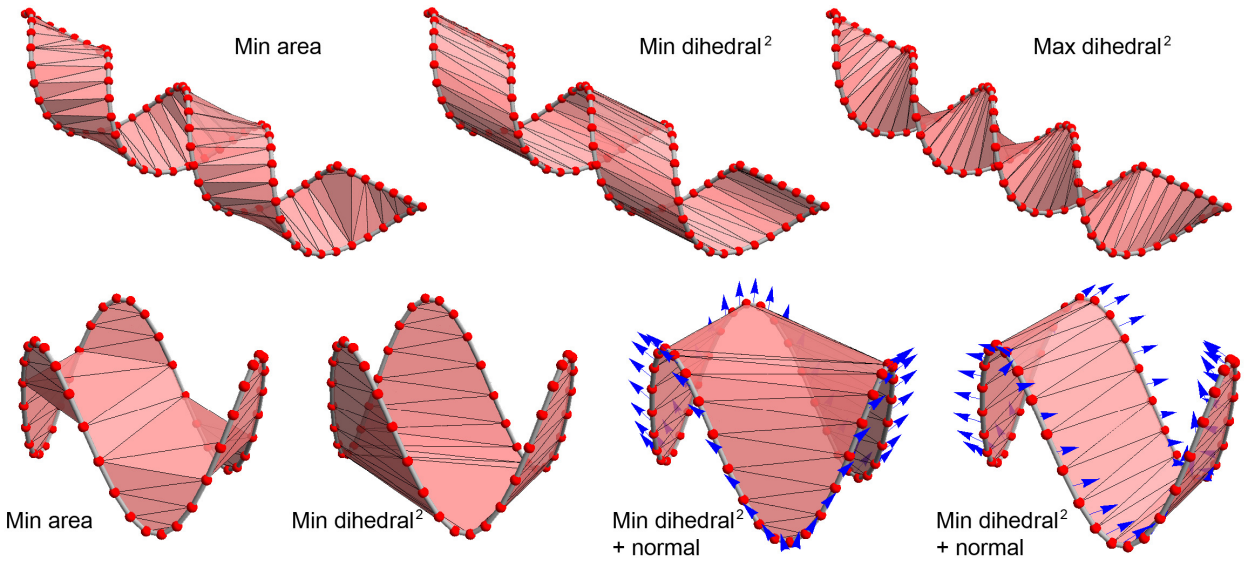

Figure 5.1: Variations of triangulations generated by our algorithm for *Spiral* (top) and *Monkey* (middle). Prescribed boundary normals, if present, are shown as arrows.

Given the association between the islands and their surrounding hole (which we did manually for this example), we triangulated each hole using Delaunay triangles. To produce hole fillers that connect naturally with the surrounding geometry, the triangulation minimizes the sum of dihedral angles both at interior edges and with existing triangles surrounding the hole (the latter can be incorporated as a bi-triangle weight on boundary edges). Triangulating all 455 holes (61 holes of which have interior islands) in this model took 40 seconds.

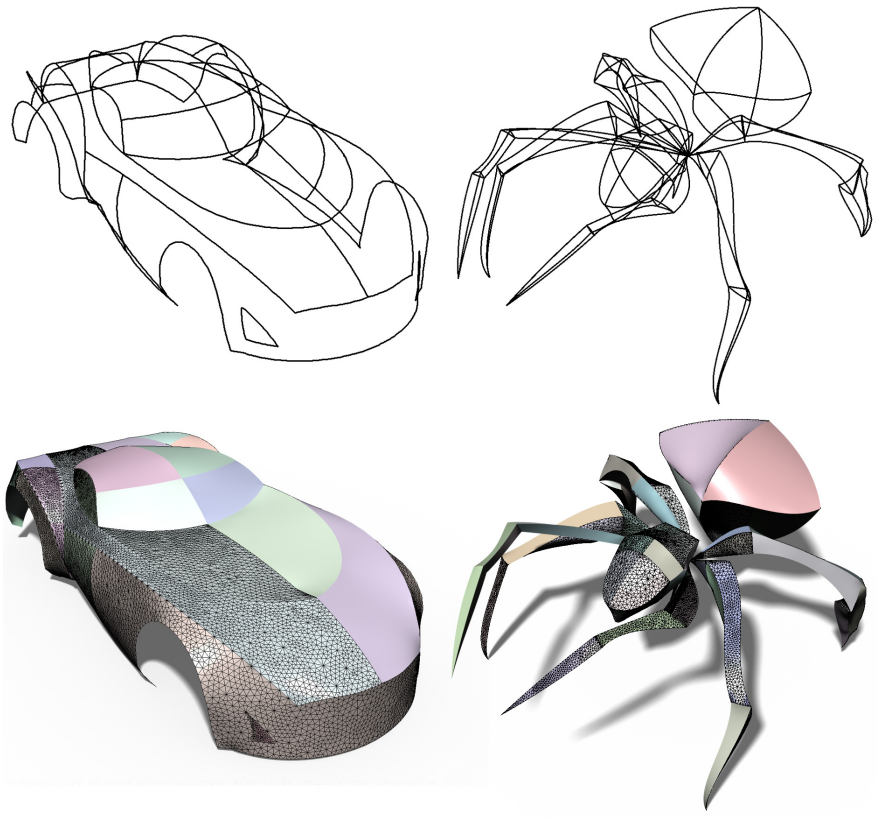

Figure 5.2: ILove Sketch curve data [5] with resolved patches [1] (top) is first triangulated minimizing the dihedral angle bending using our method. The surface is then refined using the method of [19], and a final boundary normal conforming bi-Laplacian surface is produced(shown bottom) using the method of Andrews et. al [3].

Figure 5.3: Filling a model with numerous holes. Close-ups on a portion of the model are shown at the bottom.

Chapter 6

Conclusion

In this thesis, we presented an algorithm for optimally triangulating single or multiple 3D polygons while guaranteeing the manifold topology of the output. We have also explored an efficient approximation by restricting the search space to Delaunay triangles.

6.1 Limitations

While our algorithm guarantees to return an optimal solution when given *all* triangles as the input, the complexity of the algorithm grows quickly with the number of input polygons. Restricting the search to Delaunay triangles offers a good balance between efficiency and the optimality of results, but it could risk the failure of finding a solution for some extreme cases, when the input is not Delaunay triangulable.

We use Tetgen [25] to compute the Delaunay triangules, but Tetgen only supports input that can cast a 3D volume. To triangulate polygons in one single plane, our algorithm needs to take *all* triangles as the input.

6.2 Future work

There are several interesting extensions that we would like to explore. Firstly, while triangulating with Delaunay triangles returned solutions for almost all of our test data, there exists input polygons that cannot be triangulated by Delaunay triangles. However, we have

observed and proved that these polygons are usually highly contorted and not likely to appear in practice. Nevertheless, it would be interesting to discover other input triangle spaces (such as regular triangulations or almost-Delaunay triangulations [6]) that may lead to better triangulability and optimality while maintaining the computational efficiency. Secondly, what choice of weights would deliver the "best-looking" triangulation? For a single polygon, the literature suggests to minimize a combination of triangle areas and dihedral angles [19]. However, we observed that such weighting often does not give a fair triangulation for multiple polygons. It would be interesting to see if more sophisticated weighting schemes can improve the results, and if so, how the triangulation algorithm should be adapted. Thirdly, for the hole-filling application, it would be ideal to automatically group the island boundaries as input to the triangulation algorithm. One promising idea might be to use the cost of triangulation as a means to evaluate a grouping of curves. Moreover, in this thesis, we mainly focus on closed polygons, it would be useful to generalize the algorithm to work on open polylines as well, which are also widely used in 3D sketchings.

Appendix A

Proof of Lemma 2.2.1

Proof: Let $S = S_1 \cup S_2 \cup \{t\}$. We need to show S is manifold, has a simple topology, and uses no other weak edges than E.

Manifoldness: Since E_1, E_2 are disjoint, sets $S_1, S_2, \{t\}$ share no other edges than the two edges of t other than e_D, which we denote as e_1, e_2. So the valence of edges in S are the same as those in S_1, S_2 except for e_D (valence 1) and e_1, e_2 (valence increments by 1). If e_i $(i = 1, 2)$ is a boundary edge of D, then t must be a Case II split and the corresponding sub-domain D_i must be empty, in which case the valence of e_i in S will be 1. Otherwise, if e_i is a not a boundary edge, D_i must be non-empty (or it will have duplicated edges on its boundary and hence possess no valid triangulations) and e_i must have valence 1 in S_i, hence its valence in S will be 2.

Simple topology: Clearly S is a single connected component (because S_1, S_2 are) and has $h+1$ boundaries (due to manifoldness). We need to show that there is no high-genus features (e.g., tunnels) in S. We will validate using the Euler number. A simple calculation reveals that, if S is a valid triangulation of D, the Euler number $\chi(S)$ satisfies:

$$\chi(S) = \chi(\partial D) - h + 1 \tag{A.1}$$

where $\chi(\partial D)$ is the Euler number of the boundary loop of D and h is the number of holes in D. Suppose S_1, S_2 satisfy Equation A.1 (and let their hole counts be h_1, h_2), we will show it holds for S too. We consider the two splitting cases separately. For Case I, since S adds one edge and triangle to S_1, we have $\chi(S) = \chi(S_1)$. We arrive at Equation A.1 by the fact that $\chi(\partial D) = \chi(\partial D_1) + 1$ and $h = h_1 + 1$. For Case II, we similarly have $\chi(S) = \chi(S_1) + \chi(S_2) - c$

where c is the number of common vertices shared by the boundaries of D_1 and D_2. Equation A.1 holds due to the relations $\chi(\partial D) = \chi(\partial D_1) + \chi(\partial D_2) - c + 1$ and $h = h_1 + h_2$.

Weak edges: The claim holds if we can show that any weak edge e of D is either a weak edge of D_1 or D_2, or it is not used by S_1 or S_2. This is true because a weak edge connects two ends of boundary segments, and an end of a boundary segment in D remains and end in D_1 or D_2. So e either connects two ends that appear in the same sub-domain (in which case it is a weak edge of that sub-domain), or two ends appearing in different sub-domains (in which case it cannot appear in the triangulation of either sub-domain). $\square$

Appendix B

Proof of Lemma 2.2.2

Proof: Since S is manifold and has simple topology, the remainder $S \setminus \{t\}$ is either a single edge-connected component (when v is on a hole of D) or two connected components (when v is on the boundary of D). In either case, it is easy to see that each component is manifold and has simply topology. By the same argument about weak edges in the proof of Lemma 2.2.1, E is the joint set of weak edges in each of the component triangulations, which are also disjoint due to manifoldness of S.$\square$

Appendix C

Proof of Lemma 2.2.3

Proof: We prove by induction. Suppose both clauses hold for any sub-domain of D. We show they hold for D as well.

We start with the second clause. Suppose there is some minimal set $E \in \Pi(D, W)$ where the two weights $W(D, E), \hat{W}(D, E)$ are different. Since $\hat{W}$ takes the minimum over a smaller space than W, we have $W(D, E) < \hat{W}(D, E)$. Let $W(D, E)$ be achieved by some splitting triangle t, sub-domains $\{D_1, D_2\} \in \Phi(D, t)$, and weak edge sets $\{E_1, E_2\} \in \Psi(D_1, D_2, E)$. Then at least one of E_1, E_2 is not a minimal set for the corresponding sub-domain. Without loss of generality, suppose $E_1 \notin \Pi(D_1, W)$. By definition of minimal sets, there is some weak edge set E_1' of D_1 such that $E_1' \subset E_1$ and $W(D_1, E_1') < W(D_1, E_1)$. Let the joint set of E_1' and E_2 be E', hence $W(D, E') < W(D, E)$. However, since $E' \subseteq E$, this means that either $W(D, E)$ is not minimal (if $E' = E$) or set E is not a minimal set of D, both causing contradictions.

For the first clause, we first show that if $E \in \Pi(D, W)$, then $E \in \Pi(D, \hat{W})$. Suppose otherwise, then there is a subset $E' \subset E$ such that $\hat{W}(D, E') < \hat{W}(D, E)$. Since $W(D, E') \leq \hat{W}(D, E')$, and by the second clause, we have $W(D, E') < \hat{W}(D, E) = W(D, E)$, which contradicts to $E \in \Pi(D, W)$. We next prove the other direction, also by contradiction. Suppose there is some $E \in \Pi(D, \hat{W})$ but $E \notin \Pi(D, W)$. One can show that there exists some subset $E' \subset E$ such that $E' \in \Pi(D, W)$ and $W(D, E') < W(D, E)$. By the second clause, we have $\hat{W}(D, E') = W(D, E') < W(D, E) \leq \hat{W}(D, E)$, contradicting $E \in \Pi(D, \hat{W})$. $\square$

References

[1] Fatemeh Abbasinejad, Pushkar Joshi, and Nina Amenta. Surface patches from unorganized space curves. *Comput. Graph. Forum*, 30(5):1379–1387, 2011.

[2] Nina Amenta, Marshall Bern, and David Eppstein. The crust and the β-skeleton: combinatorial curve reconstruction. *Graph. Models Image Process.*, 60(2):125–135, March 1998.

[3] James Andrews, Pushkar Joshi, and Nathan A. Carr. A linear variational system for modeling from curves. *Comput. Graph. Forum*, 30(6):1850–1861, 2011.

[4] Marco Attene, Marcel Campen, and Leif Kobbelt. Polygon mesh repairing: An application perspective. *ACM Comput. Surv.*, 45(2):15:1–15:33, March 2013.

[5] Seok-Hyung Bae, Ravin Balakrishnan, and Karan Singh. Ilovesketch: as-natural-as-possible sketching system for creating 3d curve models. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, UIST '08, pages 151–160, New York, NY, USA, 2008. ACM.

[6] Deepak Bandyopadhyay and Jack Snoeyink. Almost-delaunay simplices: Robust neighbor relations for imprecise 3d points using cgal. *Comput. Geom. Theory Appl.*, 38(1-2):4–15, September 2007.

[7] Gill Barequet, Matthew Dickerson, and David Eppstein. On triangulating three-dimensional polygons. In *Proceedings of the twelfth annual symposium on Computational geometry*, SCG '96, pages 38–47, New York, NY, USA, 1996. ACM.

[8] Gill Barequet, Michael T. Goodrich, Aya Levi-Steiner, and Dvir Steiner. Straight-skeleton based contour interpolation. *Graph. Models*, 65:323–350, 2004.

[9] Gill Barequet and Micha Sharir. Filling gaps in the boundary of a polyhedron. *Comput. Aided Geom. Des.*, 12(2):207–229, March 1995.

[10] Mikhail Bessmeltsev, Caoyu Wang, Alla Sheffer, and Karan Singh. Design-driven quadrangulation of closed 3d curves. *Transactions on Graphics (Proc. SIGGRAPH ASIA 2012)*, 31(5), 2012.

[11] Jean-Daniel Boissonnat and Pooran Memari. Shape reconstruction from unorganized cross-sections. In *SGP '07: Proceedings of the fifth Eurographics symposium on Geometry processing*, pages 89–98, 2007.

[12] Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485–524, August 1991.

[13] Herbert Edelsbrunner and Nimish R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15(3):223–241, 1996.

[14] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Commun. ACM*, 20(10):693–702, October 1977.

[15] P. D. Gilbert. New results in planar triangulations. Technical report, Urbana, Illinois: Coordinated Science Laboratory, University of Illinois, 1979.

[16] Magdalene Grantson, Christian Borgelt, and Christos Levcopoulos. Minimum weight triangulation by cutting out triangles. In *Proceedings of the 16th international conference on Algorithms and Computation*, ISAAC'05, pages 984–994, 2005.

[17] Tao Ju. Fixing geometric errors on polygonal models: a survey. *J. Comput. Sci. Technol.*, 24(1):19–29, January 2009.

[18] G.T. Klincsek. Minimal triangulations of polygonal domains. In Peter L. Hammer, editor, *Combinatorics 79*, volume 9 of *Annals of Discrete Mathematics*, pages 121 – 123. Elsevier, 1980.

[19] Peter Liepa. Filling holes in meshes. In *Symposium on Geometry Processing*, pages 200–206, 2003.

[20] Lu Liu, C. Bajaj, Joseph Deasy, Daniel A. Low, and Tao Ju. Surface reconstruction from non-parallel curve networks. *Comput. Graph. Forum*, 27(2):155–163, 2008.

[21] Kenneth Rose, Alla Sheffer, Jamie Wither, Marie-Paule Cani, and Boris Thibert. Developable surfaces from arbitrary sketched boundaries. In *Proc. Eurographics Symposium on Geometry Processing*, 2007.

[22] Gerhard Roth and Eko Wibowoo. An efficient volumetric method for building closed triangular meshes from 3-d image and point data. In *Proceedings of the conference on Graphics interface '97*, pages 173–180, Toronto, Ont., Canada, Canada, 1997. Canadian Information Processing Society.

[23] Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom.*, 1:51–64, 1991.

[24] Jonathan Richard Shewchuk. Constrained delaunay tetrahedralizations and provably good boundary recovery. In *Eleventh International Meshing Roundtable*, pages 193–204, 2002.

[25] Hang Si. Tetgen:a quality tetrahedral mesh generator and a 3d delaunay triangulator, 2009.

[26] Yixin Zhuang, Ming Zou, Nathan Carr, and Tao Ju. A general and efficient method for finding cycles in 3d curve networks. *ACM Trans. Graph.*, 32(6):180:1–180:10, November 2013.