November 2015

Skeleton Structures and Origami Design

John C. Bowers
University of Massachusetts - Amherst

SKELETON STRUCTURES AND ORIGAMI DESIGN

A Dissertation Presented

by

JOHN CHRISTOPHER BOWERS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2015

School of Computer Science

SKELETON STRUCTURES AND ORIGAMI DESIGN

A Dissertation Presented

by

JOHN CHRISTOPHER BOWERS

Approved as to style and content by:

Ileana Streinu, Chair

Andrew McGregor, Member

Gerome Miklau, Member

Tom Braden, Member

Lori Clarke, Chair
School of Computer Science

DEDICATION

For Katherine and Pippin and Scout.

ACKNOWLEDGMENTS

I would like to express my heartfelt thanks and appreciation for the many people who helped me in completing this thesis.

This thesis would not have been possible without the advice and support of my advisor, Ileana Streinu. Ileana's adept guidance was crucial to my maturing as a researcher, a scholar, and a teacher. From her I learned perseverance (especially when facing reviewers), confidence and independence, and above all how to be a researcher. When I needed it, she pushed me (especially to revise, revise, revise!), and when I needed to step up on my own, she gave me free rein to do so. Most importantly, she introduced and fostered in me the joy of computational geometry. I remember coming away from one of the first lectures in Ileana's class thinking, *this is it; this is what I want to study.* I am very grateful for her mentorship and greatly value my time working with and learning from her.

I also want to thank my committee, Andrew McGregor, Gerome Miklau, and Tom Braden, for their support and guidance. Additionally, I thank Andrew for many helpful conversations over the course of my graduate career. I thank Tom for helping me to improve the writing of this document.

I next want to thank Rui Wang, who helped me get my start as a researcher, took me to my first academic conferences, and has been an inspiration and a support throughout my graduate career. Rui's energy and enthusiasm is infectious and he is a model for the type of researcher and advisor I want to be. I hope that my interactions with future students will be as energetic and inspiring as Rui's were with me.

Especially in the last year, I have benefitted greatly from the professional advice and sound council of many that traveled this path before me. I thank Andrew Berke,

Henry Field, Jackie Field, Megan Olson, Audrey St. John, and Jerod Weinman for encouragement at the last mile, for proofreading, and especially for advice that gave me the courage to complete this thesis and start my career as a professor.

I want to thank the many Theory and LinKaGe Lab students who were in the trenches with me: Ashraf Alam, Marco Carmosino, Michael Crouch, Naomi Fox, Cibele Freire, Filip Jagodzinski, Brandon McPhail, Daniel Stubbs, Patrick Taylor, David Tench, Hoa Vu, and Sofya Vorotnikova. Through conversation, debate, whiteboard mini-lectures, practice talks, and plenty of coffee, I found encouragement, inspiration, and above all kept my sanity. A special thanks as well to Leeanne Leclerc, who deftly shepherded me (and many before me) through the many hoops and requirements of the graduate school.

A large part of this work was supported by the National Science Foundation Graduate Research Fellowship Program.

I first started down the path of a computer scientist when a friend of my parents, Steve Payne, taught me to program in sixth grade. Since then Steve has become both a mentor and a close friend. I am very much in debt to Steve for the years of support, guidance, and prayers. I remember at one point during high school a warning he gave me that I (mostly) took to heart. He warned me it sometimes becomes necessary to focus on one direction and not to simply become a Jack of All Trades and Master of None. This thesis is my heeding of that advice.

I also thank those many friends who have given me love and support throughout this thesis process; for the friendship and support of Craig Nicolson–he has been a invaluable fellow traveller in The Way and additionally helped me to grow as a presenter and speaker; for Ian Callahan, whose courage and steadfastness inspires me daily; for Mike Foster and Richard Vachet, who model the sort of professor I want to be; for Steve Oloo, who has shared the PhD journey with me; for Ben Greene, who adventured into forest, over cliff, and up ice flows, when I needed to escape the LED

glare of my computer screen; and for Nate Daman, who has been a parenting role model, friend, and council.

Above all I gratefully thank my wife, Katherine Bowers, and my family. This thesis is a part of my and Katherine's shared vocation and I thank her for her patience and support throughout the process of completing it. She is my best friend, and is a wonderful mother to our little ones. This thesis is dedicated to her and to our two children, Pippin and Scout, who are my daily joy and inspiration. I thank my parents, Phil and Kris Bowers, for the innumerable ways they have supported me and nurtured me–to Dad, especially for the many mathematical conversations and coffee supply; and to Mom, especially for the prayers and encouragement and teaching advice. I thank my siblings, Maddy and Thomas Bowers, for relaxing and recreating with me. I thank my parents-in-law, Eric and Lu Grimm, and my sisters-in-law, Kellie Bowers and Beca Grimm, for the years of encouragement, and countless ways they have made me feel supported and loved.

ABSTRACT

SKELETON STRUCTURES AND ORIGAMI DESIGN

SEPTEMBER 2015

JOHN CHRISTOPHER BOWERS

B.Sc., THE FLORIDA STATE UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Ileana Streinu

In this dissertation we study problems related to polygonal skeleton structures that have applications to computational origami. The two main structures studied are the *straight skeleton* of a simple polygon (and its generalizations to planar straight line graphs) and the *universal molecule* of a *Lang polygon*. This work builds on results completed jointly with my advisor Ileana Streinu.

Skeleton structures are used in many computational geometry algorithms. Examples include the medial axis, which has applications including shape analysis, optical character recognition, and surface reconstruction; and the Voronoi diagram, which has a wide array of applications including geographic information systems (GIS), point location data structures, motion planning, etc.

The *straight skeleton*, studied in this work, has applications in origami design, polygon interpolation, biomedical imaging, and terrain modeling, to name just a few. Though the straight skeleton has been well studied in the computational geometry

literature for over 20 years, there still exists a significant gap between the fastest algorithms for constructing it and the known lower bounds.

One contribution of this thesis is an efficient algorithm for computing the straight skeleton of a polygon, polygon with holes, or a planar straight-line graph given a secondary structure called the induced motorcycle graph.

The *universal molecule* is a generalization of the straight skeleton to certain convex polygons that have a particular relationship to a metric tree. It is used in Robert Lang's seminal TreeMaker method for origami design. Informally, the universal molecule is a subdivision of a polygon (or polygonal sheet of paper) that allows the polygon to be "folded" into a particular 3D shape with certain tree-like properties. One open problem is whether the universal molecule can be rigidly folded: given the initial flat state and a particular desired final "folded" state, is there a continuous motion between the two states that maintains the faces of the subdivision as rigid panels? A partial characterization is known: for a certain measure zero class of universal molecules there always exists such a folding motion. Another open problem is to remove the restriction of the universal molecule to convex polygons. This is of practical importance since the TreeMaker method sometimes fails to produce an output on valid input due the convexity restriction and extending the universal molecule to non-convex polygons would allow TreeMaker to work on all valid inputs. One further interesting problem is the development of faster algorithms for computing the universal molecule.

In this thesis we make the following contributions to the study of the universal molecule. We first characterize the tree-like family of surfaces that are foldable from universal molecules. In order to do this we define a new family of surfaces we call *Lang surfaces* and prove that a restricted class of these surfaces are equivalent to the universal molecules. Next, we develop and compare efficient implementations for computing the universal molecule. Then, by investigating properties of broader classes

of Lang surfaces, we arrive at a generalization of the universal molecule from convex polygons in the plane to non-convex polygons in arbitrary flat surfaces. This is of both practical and theoretical interest. The practical interest is that this work removes the case from Lang's TreeMaker method that causes TreeMaker to fail to produce output in the presence of non-convex polygons. The theoretical interest comes from the fact that our generalization encompasses more than just those surfaces that can be cut out of a sheet of paper, and pertains to polygons that cannot be lied flat in the plane without self-intersections. Finally, we identify a large class of universal molecules that are not foldable by rigid folding motions. This makes progress towards a complete characterization of the foldability of the universal molecule.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Skeleton Structures. A *skeleton* is a structure on the interior of the polygon, which is typically a (combinatorial) tree. It subdivides the polygon into faces and in a qualitative sense captures certain properties of a polygon (like its shape) in a way that is often useful in applications.

Examples of skeletons are the well-known *medial axis* [10], the *linear axis* which is obtained from and approximates the medial axis [56], *bisector graphs*, and the *straight skeleton* [4]. Each of these is a tree-like structure on the interior of a polygon and each has generalizations to other settings such as polygons with holes, or planar straight line graphs. The medial axis is composed of both straight edges and curves.

The Straight Skeleton. Arising from the need in applications to deal with straight edges only, the straight skeleton was introduced to provide a similar structure that (qualitatively) "looks like" the medial axis, but is composed of straight line segments. The straight skeleton has found many applications. To name several: roof design [4]; terrain modeling [3]; polygon interpolation [6]; graph drawing [21]; procedural modeling of urban environments [57]; biomedical imaging [26]; polygon decomposition [55]; and computational origami [27]. The straight skeleton of a polygon is illustrated in Fig. 1.1, left.

For a convex polygon the straight skeleton and the medial axis are identical. However, in the non-convex case, they differ significantly. The difference arises chiefly from the property that while the medial axis is stable under small changes in the polygon, the straight skeleton (in certain situations) is highly unstable, and small changes in

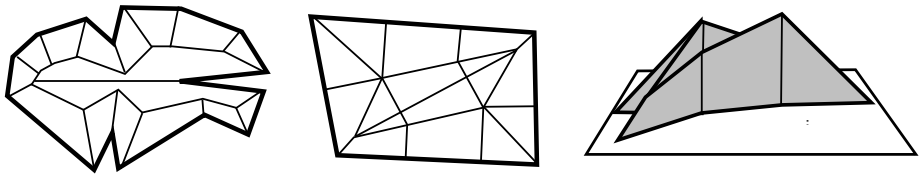


Figure 1.1: Left: the straight skeleton of a polygon. Middle: the universal molecule of a polygon. Right: a "folding" of the universal molecule in $\mathbf{R}^3$.

the polygon may result in drastic changes in the straight skeleton's structure. It appears that this contributes to the current speed gap between the fastest algorithms for computing the medial axis and straight skeleton: the medial axis of a non-convex polygon with n vertices can be computed in linear time [25], but currently the fastest known algorithms for computing the straight skeleton take $\omega(n \text{ polylog}(n))$ in the worst case.

Algorithmic complexity of the straight skeleton. Currently, the best lower bounds known for the straight skeleton problem are $\Omega(n)$ in the case of polygons, and $\Omega(n \log n)$ in the case of polygons with holes and general planar straight line graphs. For the past two decades, this gap has lead to much interest in developing faster algorithms for computing it. *A main contribution of this thesis is an $O(n \log n)$ time algorithm for computing the straight skeleton of a polygon from its induced motorcycle graph and $O(n \log n \log m)$ time algorithm for computing the straight skeleton of a planar straight line graph with m-connected components from its induced motorcycle graph.* This work has been submitted and is under review.

Origami Design. One interesting application area of the straight skeleton is that of *computational origami*. The ***origami design problem*** is a term used to denote a family of related problems. Generally, the goal is to compute a crease pattern on a sheet of paper so that when the paper is folded along the creases the result is some desired 3D shape with certain geometric properties (a description of exactly what properties are desired is specific to the particular design problem).

For example, the *fold-and-one-cut (origami design) problem* is: given a square sheet of paper and a polygon drawn on the paper, fold the paper into a shape so that a single cut from a pair of scissors cuts only along the drawn polygon's boundary, effectively "cutting out" the polygon from the paper in a single cut. This problem has been solved using straight skeletons [28]. Other notable design problems include polyhedron wrapping [29], in which the goal is to "wrap" the surface of a polyhedron with a sheet of paper; silhouette folding [29], in which the goal is to fold a square sheet of paper flat so that its silhouette is a desired 2D shape; and polyhedral folding [52], in which the goal is to cut a polygon out of a sheet of paper and fold it into a desired triangulated polyhedral surface.

Rigid folding motions. Typically a solution to an origami design problem is an algorithm that given the paper and desired constraints produces a crease pattern such that there *exists* an isometric reconfiguration of the paper into 3D that realizes the constraints. However the existence of such a reconfiguration does not of itself guarantee the existence of a folding animation, or *rigid folding motion*, that begins with the flat paper and continuously folds the paper only along creases to reach the final state without stretching or bending any faces. The problem of producing such a motion is the ***rigid folding problem***.

Lang's TreeMaker method for origami design and the universal molecule. One of the first origami design problems studied in the literature is R. Lang's TreeMaker problem [44]. The problem is: given a geometrically embedded tree T and a square sheet of paper, produce a crease pattern on the paper so that it folds into 3D in such a way that its orthogonal projection into the xy-plane is equivalent to T. Lang's solution, the TreeMaker method, works in two phases. The first phase subdivides the paper into a series of polygons. The second phase "fills in" the interior of each polygon with a particular crease pattern called the *universal molecule*. The universal molecule of a polygon, and a "folding" of it into $\mathbf{R}^3$ are depicted in Fig. 1.1 middle

and right. The universal molecule is a generalization of the straight skeleton and under certain conditions is identical with it.

Problems related to TreeMaker. One difficulty with TreeMaker is that its first phase may produce polygons which are non-convex, but the universal molecule is defined only for convex polygons. In this case TreeMaker simply fails to produce an output. A main contribution of this thesis is to generalize the universal molecule to cover all possible polygons produced by the first phase of TreeMaker (see Ch. 7). This settles an open conjecture of Demaine and O'Rourke (Conjecture 16.8.1 in [31]). This is joint work with my advisor Ileana Streinu and is under review. Though the algorithm has been in practical use for some 20 years and has been studied in various forms in the computational geometry literature, no precise mathematical characterization of its output independent of the algorithm had appeared. In order to generalize the the algorithm, we first need such a characterization, which is another contribution of this thesis, given in Ch. 5. This work has appeared in [15]. Along the way, we develop and analyze two $O(n^2 \log n)$ algorithms for computing the universal molecule (see Ch. 6). This is joint work with Ileana Streinu and appeared in [14].

A remaining problem concerns when a universal molecule has a rigid folding motion. Prior to the present work, it was known that in the highly specialized case where the universal molecule is identical to the straight skeleton, there always exists a rigid folding motion (cf. [27]). Given a particular tree, the subset of compatible polygons for which the universal molecule is the straight skeleton has measure zero. In Ch. 8 we characterize a larger class of polygons for which we show that not only does no rigid folding to the final desired state exist, but no nontrivial rigid folding from the flat state to any other state exists. This is joint work with Ileana Streinu and has appeared in [17].

CHAPTER 2

PRELIMINARIES

In the next section, we provide preliminary definitions of our main objects of interest, the straight skeleton and the universal molecule, as well as additional terminology needed throughout this thesis including the straight skeleton roof, motorcycle graphs, doubling polygons, and Lang polygons. In each chapter, to aid the reader, we review the relevant concepts and where appropriate give additional details apropos to the chapter. In Sec. 2.1 we cover some basic definitions. In Sec. 2.2 we define the straight skeleton and investigate some of its properties. In Sec. 2.3 we define the universal molecule.

2.1 Basic Definitions

Before defining the straight skeleton and the universal molecule, we establish some basic definitions.

2.1.1 Polygons

Informally, a polygon is a closed chain of straight-line segments drawn in the plane laid out end-to-end. The vertices of the polygon are the points joining consecutive line segments and the edges of the polygon are the straight line segments. This is a decent first pass at a definition, but it rules out certain situations that we would like to capture. Specifically, we would like to be able to represent polygons that have zero-length edges, and polygons that have "straight vertices", or vertices making an angle of π in the plane. In order to do this, it is helpful to separate the combinatorics from the geometry.

Combinatorial chains. It is thus helpful to start by defining a ***cycle***, which is a cyclically ordered list of abstract objects called ***vertices***. Thus, the cycle with n vertices is given by the list $1, \ldots, n$. The ***edges*** of the cycle are given by all consecutive pairs of vertices $(i, i+1)$. Since we treat the list as cyclic, the pair $(n, 1)$ is an edge in the polygon. If instead of treating the list a cyclically ordered, we simply treat it as an ordered list, we say that we have a ***chain***. As before, each pair $(i, i+1)$ is an edge of the chain, but the pair $(n, 1)$ is not. We make a cycle (or chain) ***metric*** by defining a ***weight function*** that assigns to each edge $(i, i+1)$ in the chain a real number called its ***weight***. We typically denote the weight function by w, so if $e = (i, i+1)$ is an edge of a cycle, its weight is denoted by $w(e)$. We typically denote a weighted chain as the pair (C, w).

Planar polygons. Let C be a cycle with n vertices. To obtain a ***polygon*** realizing C, we assign a point $\mathbf{p}_i$ to each vertex i in C. The polygon is defined as the geometric figure that realizes each vertex i in C as its assigned point $\mathbf{p}_i$ and each edge $(i, i+1)$ in C as the straight-line segment $\mathbf{p}_i\mathbf{p}_{i+1}$. (If C is a chain, then we call this a ***polygonal chain***.) If C is a weighted cycle, then we say that an assignment of points to the vertices of C realizes C as a polygon only if the length of each line segment $\mathbf{p}_i\mathbf{p}_{i+1}$ is equal to the weight $w((i, i+1))$. This is, of course, only possible if all of the weights are non-negative, since a line segment cannot have a negative length (we do, however, allow "line segments" between a point and itself, thus having vanishing length).

Note that this construction of a polygon explicitly allows the "oddities" we mentioned above: if $\mathbf{p}_i = \mathbf{p}_{i+1}$, then the edge $(i, i+1)$ is realized as a "zero-length edge" in the plane. Additionally, two line segments can be collinear under this construction, thus having a "straight" vertex between them. Separating the combinatorics from the realization helps us to track these sorts of features.

Abuse of terminology. In the remainder, rather than starting with a cycle C and then defining a polygon by assigning points to each vertex, we will typically define a polygon P by giving the list of its points $(\mathbf{p}_1, \ldots, \mathbf{p}_n)$. When we do this, we assume the cycle $C = (1, \ldots, n)$ is implied. We abuse the terminology slightly and refer to each point $\mathbf{p}_i$ as a ***vertex*** and each straight-line segment $\mathbf{p}_i\mathbf{p}_{i+1}$ as an ***edge***. The reader should keep in mind, however, that we always view the polygon as being first a combinatorial object that is then realized geometrically.

Simple polygons. A polygon ***self-touches*** if an edge $\mathbf{p}_i\mathbf{p}_{i+1}$ (geometrically) contains another point $\mathbf{p}_j$ (where $i, i+1 \neq j$). A polygon ***self-crosses*** if two of the segments $\mathbf{p}_i\mathbf{p}_{i+1}$ and $\mathbf{p}_j\mathbf{p}_{j+1}$ intersect on their interior. We say that a polygon is ***simple*** if it neither self-touches nor self-crosses.

Monotone polygons and chains. A polygon P is ***monotone*** with respect to a line l if P is simple and if any line l' that is orthogonal to l intersects P in at most two points. A polygonal chain is monotone with respect to a line l if it is simple and if any line l' orthogonal to l intersects the chain in at most *one* point. A monotone polygon can be divided into exactly two unique monotone chains.

Interior and exterior. A simple polygon divides the plane into a well-defined ***interior*** and ***exterior***. The two edges incident to a vertex make two angles, one on the interior and one on the exterior of the polygon, which we call the ***interior and exterior angles***. If the interior angle of a vertex is less than π, we say that that the vertex is ***convex***; equal to π then we say that the vertex is ***straight***, and if greater than π we say that the vertex is ***reflex***. A ***convex*** polygon is one in which all of its vertices are convex (or, sometimes, straight). If the polygon contains at least one reflex vertex, then it is ***non-convex***.

Orientation. Let $P = (\mathbf{p}_1, \ldots, \mathbf{p}_n)$ be a simple polygon. Geometrically, we obtain the same figure in the plane if we reverse the order on the points in P to be $(\mathbf{p}_n, \ldots, \mathbf{p}_1)$. However, these correspond to different ***orientations*** of the polygon P. Suppose we start walking around the polygon from $\mathbf{p}_1$ towards $\mathbf{p}_2$. Throughout the walk the interior of the polygon lies to the same side. If the interior is to our left, then we say the polygon is ***counter-clockwise (ccw) oriented***. Otherwise, we say that the polygon is ***clockwise (cw) oriented***. The orientation of the polygon also induces a direction on each edge from a ***source*** vertex to a ***destination***. In the remainder we assume, unless otherwise stated, that all polygons are counter-clockwise oriented.

Supporting lines. Each edge of the polygon is a straight line segment, and thus is ***supported*** by a unique line. Let $E = (e_1, \ldots, e_n)$ be the edges of a polygon P then the list $L = (l_1, \ldots, l_n)$ where each l_i is the supporting line for e_i is the list of ***supporting lines*** for P. The counter-clockwise orientation also induces an orientation of each supporting line l_i. We give to each line a well-defined ***left side*** and ***right side*** by which side the interior of the polygon lies incident to the edge e_i supported by l_i.

Defining a polygon by its supporting lines. Above we defined a polygon by starting with a cycle of points, and then derived a list of supporting lines. It is useful to also have the opposite view–start with a list of lines and then derive a cycle of points. Let $L = (l_1, \ldots, l_n)$ be a list of lines such that each pair of consecutive lines (l_i, l_{i+1}) intersect at a single point (rather than are parallel or equal). Then the polygon induced by L is given by $P = (\mathbf{p}_1, \ldots, \mathbf{p}_n)$ where each $\mathbf{p}_{i+1}$ is the intersection of lines l_i and l_{i+1}. This construction is useful in understanding the following definition of an offset polygon.

Parallel offset polygons. A parallel offset polygon is a polygon formed by starting with one polygon P and "moving" each of its edges inward in parallel by a certain

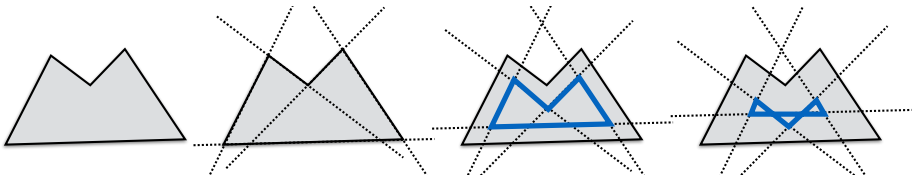


Figure 2.1: A polygon (left), its supporting lines (center left), and two offset polygons (center right and right) in blue.

amount. Figure 2.1 illustrates this concept. In this thesis, we will restrict our discussion to parallel offset polygons where each edge is moved by the same amount. We now make this more precise.

There are several ways to define a parallel offset polygon for a polygon P. We first use the supporting lines $L = (l_1, \ldots, l_n)$ of P. The ***parallel offset polygon*** of distance d is given by moving each line l_i to its left orthogonally by d units (recall that since we assume all polygons are oriented, then the left of l_i is towards the interior of P at the supported edge e_i).

An equivalent, but less evocative, way to define the parallel offset polygon is to move each vertex $\mathbf{p}_{i+1}$ of P inwards along its ***interior angle bisector***, the line through $\mathbf{p}_{i+1}$ which divides its interior angle into two equal angles. Let l_i and l_{i+1} denote the supporting lines of the two edges of P incident to $\mathbf{p}_{i+1}$. We note that all points on the interior angle bisector of $\mathbf{p}_{i+1}$ are equidistant from l_i and l_{i+1}. To form the parallel offset polygon of P of distance d we move $\mathbf{p}_{i+1}$ inwards along its interior angle bisector to the point $\mathbf{p}'$ that is a distance of d from l_i and l_{i+1}. From elementary trigonometry it follows that $\mathbf{p}'$ is the point $d/\sin(\theta/2)$ units inward along the interior angle bisector (where θ denotes the interior angle measure).

Offset polygons need not be simple. We note that an offset polygon need not be simple as is the case in the left of Figure 2.1. In that figure, though the starting polygon is simple, the offset polygon, shown in blue, has two self-crossings. Another technical fact (which is not needed in this thesis, but is interesting) if orient the

polygon in Figure 2.1 ccw, we can obtain a simple cw oriented polygon by choosing a large enough offset distance.

Parameterizing the offset polygons. We now parametrize the offset polygons defined above by letting $P(t)$ denote the offset polygon of distance t from P. This parametrization will be important for defining both the straight skeleton and the universal molecule. We can view the set of offset polygons $P(t)$ as a motion starting with the initial polygon at time $t = 0$. As t increases continuously, the lines supporting the edges of the ***sweep polygon*** move inwards in parallel at constant speed.

Events. For small enough values of t the topology of the offset polygon $P(t)$ is the same as the initial polygon P–i.e. if the polygon is simple, then for small t so is $P(t)$; however, the topology changes at certain discrete times, called ***events***.

The first event is an ***edge collapse***, where one of the edges of the polygon shrinks to zero length. If we continue the offsetting process past this event, then $P(t)$ is no longer simple. In our supporting lines based definition of the offset polygon, this means that three (or more) consecutive supporting lines intersect at the same point. In our vertex-moving based definition, this means that two consecutive vertices have the same coordinates in $P(t)$.

The second event is a ***collision*** event, where a self-touching is introduced in $P(t)$. In other words, this occurs when a point "hits" an edge elsewhere in the polygon. As with an edge collapse, if we continue past this event, then the polygon is no longer simple. If the polygon is simple leading up to such an event, then the colliding vertex is reflex.

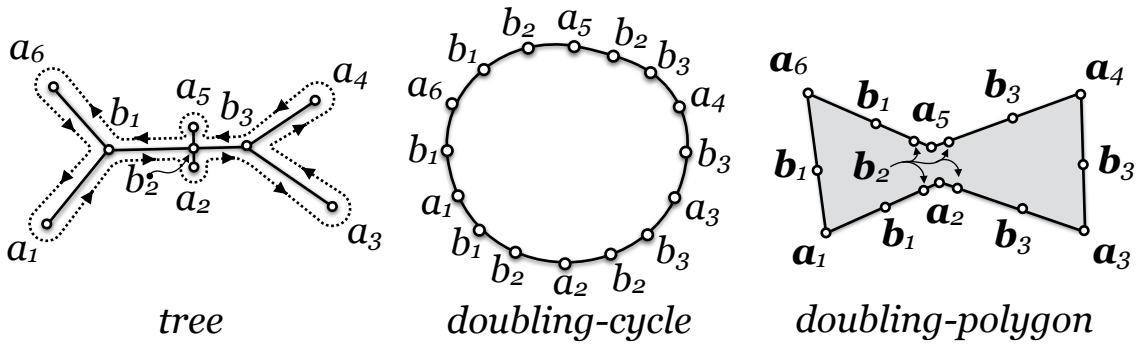


Figure 2.2: A tree, doubling cycle, and doubling-polygon.

2.1.2 Metric trees, doubling cycles, and doubling polygons

Here we define metric trees, doubling cycles, and doubling polygons. We also define a splitting operation on each. These are required for the definition of the universal molecule.

Metric trees. A (positively weighted) ***metric tree*** (T, w) is a tree T and a weight function w that maps each arc[1] of T to a positive weight or ***length***. We assume that a cyclic ordering, or rotation, is given for the incident arcs at each node[2]. Figure 2.2 (left) depicts an embedding of a weighted, topologically embedded tree with 6 leaf nodes and 3 internal nodes. The ordering of the arcs at each internal node is given by the counter-clockwise ordering depicted in the drawing. The weight of each arc is given by the length of the arc in the drawing.

Doubling cycles. If we start at a_1, and begin a walk around the tree in which we always make the right-most turn each time we reach a node, then the resulting walk encounters the nodes in the following order (a_1,b_1,b_2, a_2, b_3, a_3, b_3, a_4, b_3, b_2, a_5, b_2, b_1, a_6, b_1, a_1). This list is a cycle on the tree, which we call its ***doubling cycle*** C_T

[1]To avoid confusion, we use the terms ***node*** and ***arc*** to refer to the elements of a tree, and ***vertex*** and ***edge*** to refer to the elements of a polygon or embedded straight-line graph.

[2]Such a tree is sometimes called a ribbon tree, or a topologically embedded tree.

(cf. the center of Fig. 2.2). The reader should note that the doubling cycle for a tree T is unique, in the sense that treated as a cyclic list, the result is the same no matter which leaf node we start from. The doubling cycle C_T has the property that each node of the tree T appears as many times in C_T as is equal to its degree in T. We call each pair of two consecutive nodes in a doubling cycle an ***edge***. Each arc of the tree appears as exactly two edges in the doubling cycle, once in each direction. If we further define a weight function w on the edges of C_T, that assigns to each edge in C_T the same weight as that of the corresponding arc in T, then we say that (C_T, w) is ***the weighted doubling cycle*** for T.

Doubling polygons. Finally, if a weighted doubling cycle (C_T, w) is realized as a polygon P_T, such that each edge of the polygon has length equal to the corresponding weight in (C_T, w), then we say that P_T is a ***doubling polygon*** for T.

Notation. It is convenient to separate the n leaf nodes and m internal nodes of a tree T into two sets $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$, respectively. In order to make clear the correspondences between a tree T and a doubling polygon P_T, we use **bold face** to denote vertices of the polygon and *italics* to denote corresponding nodes in the tree. For instance, the vertex $\mathbf{a}$ in P_T corresponds to the leaf node a in T and the edge $\mathbf{ab}$ corresponds to the leaf arc ab.

Splitting trees, cycles, and polygons. Given an embedded tree T and two leaf nodes a_i and a_j, the ***splitting operation*** returns two trees T_1 and T_2 corresponding to the part of the tree to the left of (and including) the path from a_i to a_j in T, and the part to the right (resp). To split a doubling cycle C_T between a_i and a_j, we first split C_T into two open chains C_1 and C_2, one from a_i to a_j and the other from a_j back to a_i. We then close each chain using a copy of the path from a_i to a_j in T. The chains C_1 and C_2 are then doubling cycles for T_1 and T_2 (resp). In a doubling polygon P_T we allow this operation only if the shortest path between $\mathbf{a}_i$ and $\mathbf{a}_j$ is a straight-

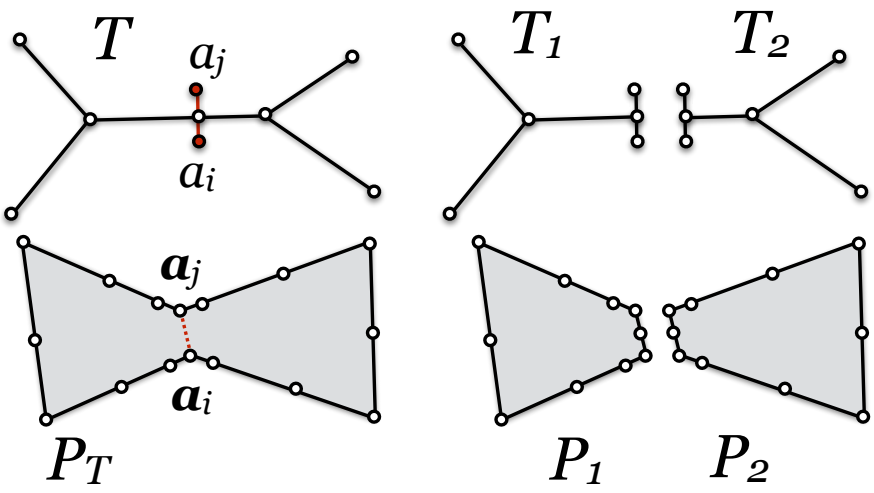


Figure 2.3: Splitting a tree and corresponding doubling polygon between a_i and a_j.

line segment (in which case we call $(\mathbf{a}_i, \mathbf{a}_j)$ a ***visible pair***), and the length of the segment is equal to $d_T(a_i, a_j)$. The split in the polygon is performed by introducing a ***splitting edge*** along the shortest path between $\mathbf{a}_i$ and $\mathbf{a}_j$ and subdividing it into edges so that it is metrically and combinatorially equivalent to the path between a_i and a_j in the tree. See Fig. 2.3.

2.1.3 Piecewise linear metric surfaces and terrains

Though both the straight skeleton and the universal molecule live naturally on the interior of polygons in the plane, it is convenient to work with them instead as piecewise linear metric surfaces. A ***piecewise linear metric surface*** (hence ***surface***) is obtained by gluing flat, polygonal faces together along whole edges. It is best to think of these as being constructed by a cookie cutter method. Each face is "cut out" of some plane and then faces are glued together in order to form more complex surfaces. Note that we "forget", as it were, any particular situating of each face in $\mathbf{R}^3$. We remember for each face only its local geometry and how it is glued together to the other faces to form a surface. A ***realization*** of a surface is a map taking each vertex to a point in $\mathbf{R}^3$, each edge to a straight-line segment, and each

face to a flat (meaning planar) polygon in $\mathbf{R}^3$ such that the edges and faces maintain their size and shape.

Intrinsic vs. extrinsic properties The description of a surface above may seem mystifying at first. What exactly is the point of forgetting how each face was cut out? Why not just define a surface as usual as some part of $\mathbf{R}^3$? One reason for doing this is to model what happens as an origami shape is folded. Suppose we fold an origami shape out of a sheet of paper. Our origami shape in $\mathbf{R}^3$ has, in one sense, a very different geometry than the original flat sheet of paper. However, from the perspective of a flatlander (someone confined to live on the paper's surface, say a bacterium) living on the surface of the origami, nothing has changed. For instance, suppose we are asked the distance a flatlander would have to travel between two points p and q. It turns out that the shortest path between p and q on the folded origami is precisely along the folded version of the straight line segment connecting p and q on the flat piece of paper. In other words, to tell the distance between p and q in the folded origami we can simply unfold the paper and measure the distance from p to q with a straight ruler. So the distance between p and q is somehow intrinsic to the paper and not a feature that is changed by folding. Given this discussion, it should be somewhat intuitive that both the flat and the folded paper are really the same surface even though certain geometrical properties have changed in the ambient space $\mathbf{R}^3$. This is the essential difference between intrinsic and extrinsic geometry, and the purpose of the definition above is to provide an intrinsic way of defining a surface.

Properties of a surface that are true in any realization are ***intrinsic***, while those that depend on a particular realization are ***extrinsic***. This distinction is particularly important for our purposes, because two different foldings of the same origami crease pattern are intrinsically the same surface but differ in their extrinsic properties (such as the dihedral or "folding" angle between faces). Showing that a surface is a folding of

another amounts to showing that the two surfaces only differ extrinsically. Important intrinsic properties include the surface's:

- *topology*, which in this paper is either a disk (***disk-like***) or an annulus (***ring-like***); since these are the only two topologies we consider, each edge is either incident to exactly one face (a ***boundary edge***), or to two faces (an ***interior edge***);
- *(intrinsic) curvature* of the surface at a vertex (defined in the next paragraph); and
- the *geodesic distance* between two points on the surface (defined shortly).

Examples of an extrinsic properties include the *dihedral angle* between two faces at an edge, the coordinates of the vertices of the surface in $\mathbf{R}^3$, etc.

Curvature. Since our surfaces are piecewise linear, the curvature is concentrated at the vertices. A vertex has a ***face angle*** in each of its incident faces, and its ***angle sum*** is the sum over all its face angles. The (intrinsic Gaussian) ***curvature*** at a vertex is given by 2π minus its angle sum. If every internal vertex of a surface has zero curvature then the surface is (intrinsically) ***flat***, which does not require that it be realized in a single plane. A realization of a flat surface in which the dihedral angles at all interior edges is π is an ***open, flat realization***. In these terms, both the initial crease pattern drawn on the paper and the final folding of the origami are (intrinsically) flat, but only the first is in an open, flat realization. If for a given surface there exists an open, flat realization, then we say that the surface is ***flattenable***. Flattenability implies that the surface is (intrinsically) flat. The converse is true for all disk-like surfaces, but not for all ring-like surfaces. For instance, if one removes the top and bottom face from a cube, the resulting ring-like surface is flat (its curvature is zero everywhere), but it is not flattenable, since it has no open, flat realization.

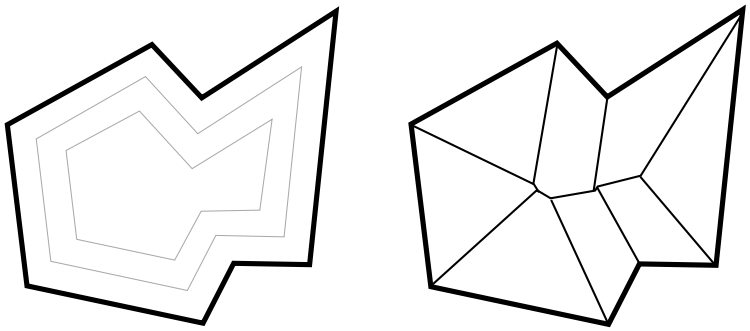
Geodesic distances and visible pairs Given a surface S, the ***geodesic distance*** between two points p and q, denoted $d_S(p, q)$, is the length of the shortest path between them, called the ***geodesic path***. On a piecewise linear surface, this is a polygonal chain and if the surface is a disk, is unique. If the geodesic path between two points p and q is (intrinsically) straight we say that (p, q) is a ***visible pair***. By this we mean that the angle made by the path in the surface is π at all points. Note that the geodesic distance $d_S(p, q)$ satisfies the usual triangle inequality–for all p, q, r, $d_S(p, q) \leq d_S(p, r) + d_S(r, q)$.

2.1.4 Lower envelopes

One geometric tool that is used in certain definitions of the straight skeleton is the lower envelope. Colloquially, the ***lower envelope*** of a set of geometric objects is the part of each object that is visible to an observer standing infinitely below the set. Lower envelopes are well studied in the computational geometry literature. We review here two concrete lower envelopes and the results relevant to our study.

Lower envelope of line segments. Concretely, let S be a set of line segments in the plane. Let $p = (p_x, p_y)$ be a point on a line segment $s \in S$. The point p is on the lower envelope of S if for all other points $q = (q_x, q_y)$ on all line segments $s' \in S$ where $p_x = q_x$ we have that $p_y \leq q_y$. In other words, the lower envelope is the set of points in S such that no other point in S lies below it. Colloquially, this can be thought of as the parts of each line segment that are visible to an observer infinitely below S (i.e. at $(0, -\infty)$. Given a segment $s \in S$, the set of points of s that lie on the lower envelope may be disconnected. In fact, the subsets of s that lie on the lower envelope are themselves line segments. The ***combinatorial complexity*** of the lower envelope is the number of line segments that appear on the lower envelope.

Given n line segments the lower envelope can be found in $O(n \log n)$ time [36] using standard line sweep techniques. Similar techniques exist for computing the



(a) (b)

Figure 2.4: (a) A polygon and two states of the straight skeleton parallel sweep, or wavefront process, on its interior. (b) The straight skeleton of a polygon.

lower envelope of a set of lines, and it is easy to extend these techniques to compute lower envelopes of a collection of both rays and line segments.

Lower envelope of slabs. A more interesting case is that of the lower envelope of slabs. For our purposes we define a slab as follows. Let Π be a plane in $\mathbf{R}^3$, $\vec{v}$ denote a vector that is parallel to Π, and C be a polygonal chain of line segments in Π that are monotone with respect to $\vec{v}$. Then the ***slab*** defined by Π, C, and $\vec{v}$ is the set of points lying "above" C in the direction of $\vec{v}$. In other words, the slab is the set of points $\{p + t\vec{v} : t \geq 0 \wedge p \in C\}$. These are sometimes thought of as polygons with vertices at infinity. As before, the lower envelope of a set of slabs is the part of each slab that is visible to an observer sitting infinitely below (meaning at $(0, 0, -\infty)$) the arrangement. Unlike the lower envelope of line segments, where the worst-case combinatorial complexity is almost linear, in the case of slabs it is super-quadratic (cf. [32]).

2.2 The straight skeleton

The ***straight skeleton*** of a polygon is an embedded straight-line graph on the interior of the polygon defined by a ***wavefront*** or ***parallel sweep process***. Move each edge of the polygon inwards at unit speed so that it remains parallel to its initial position

and trace the path of its vertices (as in the parametrized offset process in Sec. 2.1.1. See Figure 2.4a. As each edge moves, it grows or shrinks to maintain incidence with its neighbors. We start the offsetting process at time $t = 0$, and "play" the motion by allowing t to increase continuously. Recall that during the parametrized offset process, at certain discrete events, an edge may shrink to zero length (what we called an edge collapse event), or a reflex vertex may hit another edge of the polygon (what we called, in that context, a collision event). If the offsetting process is continued past such an event its topology changes, and the polygon $P(t)$ is no longer simple. To define the straight skeleton, however, we modify the polygon at each event to avoid the polygon becoming non-simple. At an edge collapse event, we remove the zero-length edge in the polygon by replacing it with a single vertex. This operation– replacing an edge with a single vertex is often referred to as a contraction, and so we call this event in the context of the straight skeleton a ***contraction event***. We then continue recursively by offsetting from the contracted polygon. At a collision event between a vertex and an edge, we split the polygon into two. In the context of the straight skeleton we call this a ***splitting event***. Once the polygon is split, we recursively continue in parallel offsetting motions simultaneously in each polygon.

Thus, we get a nested tree of parametrized offset motions. Each one starts at a simple polygon, continues until it encounters an event, processes the event by contracting or splitting, and then recursively continues on the resulting polygons. At any given time t, we have a family of polygons which are currently in motion. If one of the offset polygons contracts to a single point, then we simply remove it. We call this the a ***wavefront*** and the individual polygons ***wavefront polygons***.

Since the wavefront always moves towards the interior of each of the wavefront polygons, and we always split when a collision is encountered and contract when an edge collapses (and thus the polygons remain simple), a point on the interior of the polygon is encountered exactly once.

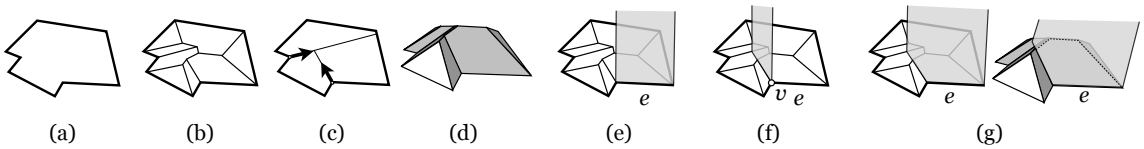


Figure 2.5: (a) A polygon. (b) Its straight skeleton. (c) The induced motorcycle graph. (d) The straight skeleton roof. (e) An edge slab. (f) The motorcycle slab for v with respect to e. (g) Shows a view of slab(e), which is the union of the edge and motorcycle slabs for e from $z = +\infty$ (left) and in perspective (right).

The straight skeleton We call the entire wavefront process the ***straight skeleton parallel sweep***. The trace of the vertices of the of the wavefront is called the ***straight skeleton***. A straight skeleton is shown in Figure 2.4b.

Properties of the straight skeleton of a polygon. The following properties of the straight skeleton were derived in [4]:

Lemma 2.2.1 ([4])**.** *Let P be a polygon with n vertices and $SS(P)$ be its straight skeleton.*

1. *$SS(P)$ is a tree on the interior of P Its leaf nodes are the vertices of P. Its arcs are straight line segments. Each internal node is either the point of an edge collapse at a contraction event or a collision at a splitting event.*

2. *The number of events encountered in the straight skeleton parallel sweep is $O(n)$.*

3. *The straight skeleton subdivides the interior of P into n faces, one for each edge of P, which we call its **base**. Each face is incident to P only along its entire base edge. Each face is monotone with respect to the line supporting its base edge.*

4. *The trace of an edge e of P (which is followed, in the event that the edge splits, in both split edges) is the face having e as its base edge.*

2.2.1 The Roof Model

A useful alternative characterization of the straight skeleton is the ***straight skeleton roof***, which is a terrain given by lifting each vertex of the straight skeleton into $\mathbf{R}^3$. An example is shown in Fig. 1.1(d). To obtain the ***straight skeleton roof*** from the straight skeleton, take each internal node of the straight skeleton and "lift" it into $\mathbf{R}^3$, by augmenting it with a z-coordinate equal to the time at which the parallel sweep encountered the node. This is equivalent to performing the sweep not in the xy-plane, but in a plane that sweeps upwards at unit speed simultaneously to the parallel sweep. In other words, at time t we lift the parallel sweep up onto the $z = t$ plane. The motion of each edge throughout the sweep is linear, and so again each edge traces out a planar polygon, this time embedded in a plane making an angle of $\pi/4$ with the face's base edge (i.e. with slope 1). We call the resulting object the ***straight skeleton roof***. It is a polyhedral surface, topologically a disk, and the projection onto the xy-plane of its vertices, edges, and faces gives the vertices, edges, and faces of the straight skeleton.

Properties of the straight skeleton roof. The straight skeleton roof has several interesting properties that were investigated in [4]:

Lemma 2.2.2 ([4])**.** *Let P denote a polygon and $R(P)$ denote the straight skeleton roof for P. Then,*

1. *Each edge e traced by a convex vertex of the wavefront in the parallel sweep forms a **ridge** in $R(P)$ (its convex dihedral angle opens downwards).*

2. *Each edge e traced by a reflex vertex of the wavefront in the parallel sweep forms a **valley** in $R(P)$ (its convex dihedral angle opens upwards).*

3. *The **descent path**, or the path of steepest descent from a point p on a face f of $R(P)$ is either a line segment joining p to the base edge of f along the slope*

of f; or, it is a polygonal path which starts by a line segment from p to a valley of f along the slope of f, which then follows the valley down to the base edge[3].

The third property in Lemma 2.2.2 might evocatively be called the ***rain-water property***, since it implies that rain falling on a particular face of the roof stays on that face until it leaves the roof at some point along the face's base edge. For this reason the straight skeleton has been applied to roof design, since for any polygon it computes a roof covering the polygon where rain water does not pool (cf. [4]).

Use in computation. As we will see in Chs. 3 & 4, the roof model proves useful in computing the straight skeleton. In fact, both the prior fastest algorithms and the algorithms presented in this thesis produce the straight skeleton roof as output. Of course, the definition of the roof model we gave above cannot be so used, because the definition assumes prior knowledge of the straight skeleton. Thus, we seek some alternative definition of the roof that does not require knowing the straight skeleton or simulating the parallel sweep. Such a definition exists, but before we can present it, we need one further object, which is called the motorcycle graph, which we now define.

2.2.2 The motorcycle graph

The main difficulty in computing the straight skeleton arises from detecting when the wavefront should be split. Such events always occur between a ***reflex vertex*** of the wavefront (a vertex whose interior angle is greater than π) and an edge. Informally, the difficulty arises from the fact that one moving reflex vertex can "cut off" another reflex vertex, which effectively separates the second from interacting with edges on the other side of the polygon. Because of this the order of events is highly

[3]In degenerate cases this may involve a chain of valley edges incident to f–more on that in Ch. 4

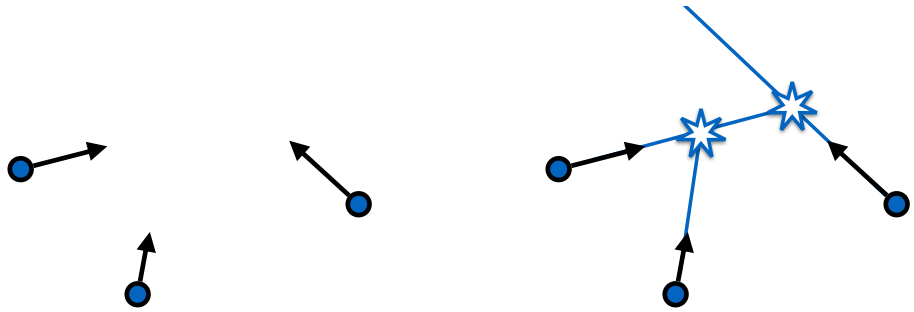


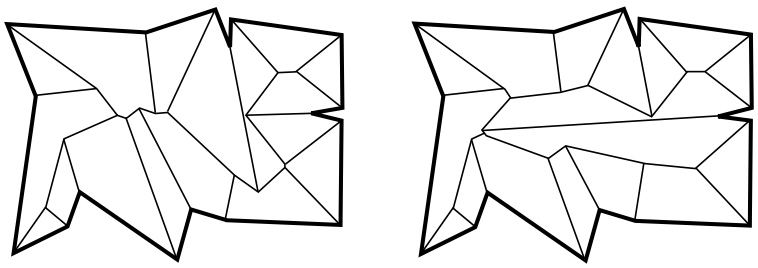
Figure 2.6: *Left:* A set of motorcycles with velocity vectors. *Right:* The motorcycle graph. One motorcycle escapes, the other two crash (at the star vertices).

susceptible to small changes in the input polygon. An example where a slight change in the polygon drastically changes the crash events is shown in Figure. 2.7.

The motorcycle graph This difficulty is modeled by an object called the ***motorcycle graph***[4]. Place a number of "motorcycles" at distinct points in the plane, each with a linear velocity. Each motorcycle moves linearly leaving a track behind it as it goes. If one motorcycle encounters the track of another then it crashes. A motorcycle that does not crash (as time goes to infinity) escapes. The motorcycle graph is given by the tracks of all motorcycles after all motorcycles have either crashed or escaped. (Note that to make this precise, we either need to add vertices *at infinity*, or compute a bounding box of all the intersections of the lines supporting each motorcycle's trajectory. If a motorcycle encounters the bounding box, then it has escaped, since no other motorcycles trajectory crosses any point of its future path.)

Motorcycle graph induced by a polygon. The ***motorcycle graph induced by a polygon*** is given by placing a motorcycle at each reflex vertex with a velocity defined so that it moves at the same speed and direction as the reflex vertex in the straight skeleton parallel sweep. Figure 2.5c depicts an induced motorcycle graph for

[4]The term "motorcycle graph" comes from its resemblance to the lightcycles in the Disney movie *Tron*.



(a) (b)

Figure 2.7: An example of a slight change in the polygon affecting a drastic change in the straight skeleton. The difference between (a) and (b) is simply a slight change in the sharpness of the reflex vertex along the right side of the polygon. The change is almost unnoticeable but drastically changes the resulting straight skeleton. Notice that not only does this significantly change the structure of the straight skeleton, but also has highly non-local effects, including changes to the combinatorics of faces whose base edges are half-way around the polygon from the changed vertex.

a polygon. A motorcycle crashes if either (a) it hits the track of another motorcycle, as before, or (b) hits some edge of the polygon.

Lifted motorcycle graph. Recall that we defined the straight skeleton roof by lifting the nodes of the straight skeleton into $\mathbf{R}^3$. Similarly, we define the ***lifted motorcycle graph*** by lifting each point on the graph so that its z-coordinate is equal to the time the corresponding motorcycle is at that point.

2.2.3 A non-procedural definition of the straight skeleton

We now give a definition of the straight skeleton roof that does not rely on prior knowledge of the straight skeleton, or a simulation of the parallel sweep. Instead, it is defined as the lower envelope of a set of partially infinite strips in $\mathbf{R}^3$ called *slabs*; however, to define the slab set requires prior knowledge of the motorcycle graph. A consequence of this definition is to break the straight skeleton problem into two pieces. First, we need an algorithm for computing the motorcycle graph of a set of motorcycles. From this, in linear time, we can compute the set of slabs. Then, once the slabs are known, we need to compute the lower envelope of the slab set. We have already noted that in general, such lower envelopes may be super-quadratic; however,

in our particular case, properties of the straight skeleton roof will prove to be useful in designing sub-quadratic algorithms for computing the lower envelope.

Slabs. For each edge of the polygon we define one ***edge slab*** and at most two ***motorcycle slabs***. A slab for an edge is defined on the plane through the edge making an angle of $\pi/4$ with the xy-plane as the area "above" the edge and its lifted motorcycle graph edges (resp)[5]. Edge and motorcycle slabs are illustrated in Figure 2.5(f, g). The union of an edge's edge slab and motorcycle slabs is its ***slab***. We give a more precise definition of this in Sec. 4.2.

Characterizing the straight skeleton roof using slabs. The ***lower envelope*** of a set of slabs is an arrangement formed by retaining only the parts of each slab that *lie below* all the other slabs, meaning a point p of a slab s is part of the lower envelope if and only if for all points q of all slabs s' such that $p_x = q_x$ and $p_y = q_y$, we have that $p_z \leq q_z$ (here p_x, p_y, and p_z denote the x, y, and z-coordinates respectively). It is helpful to think of the lower envelope as the view from $-\infty$. The following theorem characterizes the roof $R(P)$ using the slabs defined above:

Theorem 2.2.1 ([23])**.** *The straight skeleton roof for a polygon is the restriction of the lower envelope of its slabs to the region above the polygon.*

In fact, this theorem is generalized in [23] to polygons with polygonal holes. It gives a non-procedural definition of the straight-skeleton, and has, as we will see in Ch. 3, given rise to several methods for computing it which do not require the wavefront model.

Local lower envelope of a face. Theorem 2.2.1 states that the entire roof is equal to a lower envelope of slabs in $\mathbf{R}^3$. Each face also has a "local" characterization

[5]Here "above" means upwards along the slope vector of the plane.

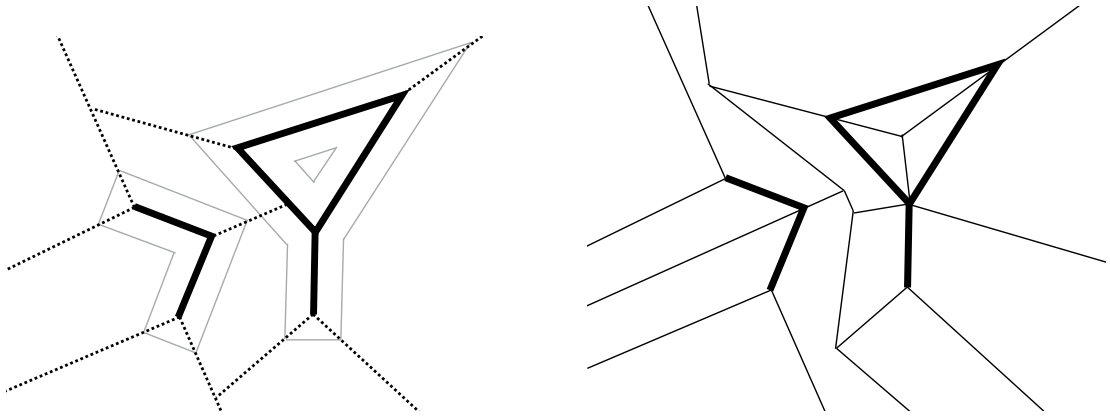


Figure 2.8: (a) A simple PSLG (thick black lines), its induced motorcycle graph (dotted lines) and a snapshot of its wavefront (thin gray lines). Note the squared caps in the wavefront emanating and two motorcycles emanating from each such vertex. (b) Its straight skeleton.

that is useful in some of the proofs in Ch. 4 . Let s_e be the slab for an edge e. If we intersect s_e with any other slab, the intersection (if it exists) is a straight line segment contained in s_e. Do this for all slabs to obtain a set of $n-1$ line segments on s_e. Now, take the lower envelope of these line segments *within* s_e with respect to the slope of s_e. In other words, we are taking the usual planar lower envelope of a set of line segments, but within the slab. The resulting chain of edges is incident to the base or motorcycle edges of the slab, and the region of the slab below this chain is equal to the face of the straight skeleton roof supported by s_e.

2.2.4 Generalizing to planar straight line graphs

Straight skeletons are also defined for the more general ***planar straight line graphs*** (PSLGs). A PSLG is a planar graph embedded in the plane so that it exhibits no self crossing nor self touching. The main difference between the polygon case and the case of PSLGs is the presence of degree 1 vertices. The wavefront definition and roof model are essentially the same as in the polygon case, except that each edge of a PSLG produces two edges in the wavefront, one on either side. Additionally, the degree one vertices are modeled by a zero-length edge in the initial

wavefront which is orthogonal to both its incident wavefront edges. This creates a *squared cap* around each degree one vertex in the wavefront. See Figure 2.8a. This is modeled in the induced motorcycle graph by shooting out two motorcycles from each degree one vertex, one along each ray making an angle of $3\pi/4$ with adjacent edge. See Figure 2.8a. Now, in addition to each edge having defined slabs, each degree one vertex has a defined ***vertex slab*** which is the union of two motorcycle slabs. Theorem 2.2.1 generalizes to:

Theorem 2.2.2 ([40])**.** *The straight skeleton roof for a planar straight line graph is the lower envelope of its edge, motorcycle, and vertex slabs.*

Figure 2.8b shows the straight skeleton of a PSLG.

2.2.5 Degenerate straight skeletons

So far, for both the straight skeleton and the motorcycle graph, we have implicitly assumed that certain ***degenerate*** situations do not arise. For instance, what do you do if two (or more) motorcycles crash into each other simultaneously? Similarly, what do you do if multiple reflex vertices collide at the same point? Another degeneracy that presents problems for some algorithms (for technical reasons) is when four slabs intersect at the same point. These situations require careful handling, and in Ch. 4 we give them a thorough treatment.

2.3 The universal molecule

Overview. The ***universal molecule*** is a particular ***origami crease pattern***, which is a planar straight line graph drawn on the interior of certain polygons, which subdivides the polygon into a piecewise linear surface. It is defined for any metric tree T and compatible convex polygon P_T, called a Lang polygon, meeting certain constraints that come from the tree. For any embedding of the tree into the xy plane, the induced piecewise linear surface has a particular realization in $\mathbf{R}^3$ called

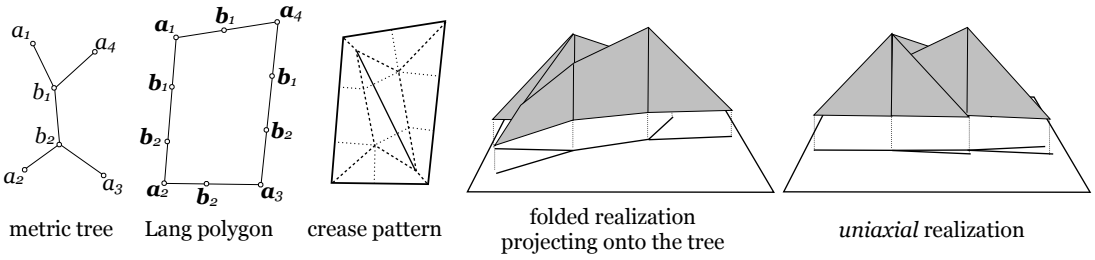


Figure 2.9: A metric tree and compatible Lang polygon, its universal molecule crease pattern, a folded realization of the crease pattern projecting onto the tree, and a realization of the crease pattern in a uniaxial state.

its ***folded realization*** which projects onto the tree. The main distinguishing feature is that it has a family of folded realizations given by embeddings of the tree into the xy-plane, so that each realization projects onto an embedding of the tree. Figure 2.9 shows an illustration of a metric tree, compatible Lang polygon, universal molecule crease pattern, and two realizations of the crease pattern in $\mathbf{R}^3$. The second is called ***uniaxial*** because all of the arcs of the tree are embedded along a common line, and the boundary of the polygon is similarly folded so that each boundary edge lies along a common line in $\mathbf{R}^3$.

Let T be a (positively weighted) ***metric tree*** (meaning a connected acyclic graph with a positive real weight $w(a)$ attached to each arc[6] a of T). We assume that each tree T is ***topologically embedded***, by which we mean that an ordering (or rotation) of the incident arcs is defined for each internal node. A ***doubling cycle*** of T is a combinatorial polygon which is a circuit on T. It is defined by starting at any leaf node and walking along a circuit of the tree while respecting the ordering on its arcs. Such a walk visits each arc exactly twice, and each node a number of times equal to its degree. Figure 2.10 illustrates these concepts. A ***doubling polygon*** is a realization of a doubling cycle in the plane which maintains the length of each edge.

[6]Note that in order to differentiate between the elements of a polygon and the elements of a tree, we will refer to *vertices* and *edges* of a polygon and *nodes* and *arcs* of a tree.

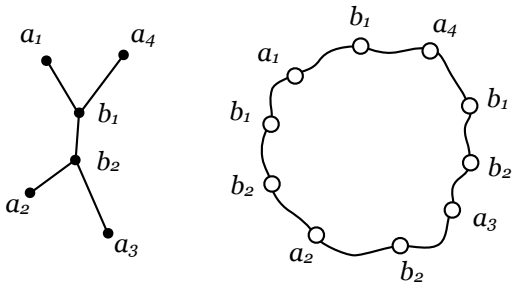


Figure 2.10: A tree (left) and a (combinatorial) doubling cycle (right) for the tree. Notice that each leaf node appears once in the doubling cycle, but the interior nodes of the tree have multiple copies.

Lang polygons A ***Lang polygon*** (T, P_T) is a pair of a (positively weighted) metric tree T and a convex doubling polygon P_T for T that satisfies the ***Lang property***: letting $\mathbf{u}$ and $\mathbf{v}$ be any two non-consecutive vertices of P_T and u and v be their respective nodes in T, we have $d(\mathbf{u}, \mathbf{v}) \geq d_T(u, v)$ (where $d(\cdot)$ is the usual euclidean distance in the plane, and $d_T(\cdot)$ denotes the distance metric on T).

The universal molecule of a Lang polygon. The universal molecule for a Lang polygon (T, P_T) is defined using a parallel sweep process similar to that of the straight skeleton. As with the straight skeleton, the ***universal molecule parallel sweep*** employs a recursive set of parametrized offset polygons. Because the polygon is convex, the only possible topological event encountered is an edge collapse. As with the straight skeleton, this is handled by contracting an edge. Unlike the straight skeleton, however, the parallel sweep is mirrored by a simultaneous ***shrinking process*** in the tree. The length of each leaf arc shrinks at the same rate as its corresponding edges in the wavefront. This maintains the invariant that the wavefront polygon is a Lang polygon for the shrinking tree except at certain event points. Maintaining this invariant past these events requires that a splitting operation be applied to both the tree and the wavefront, which splits the tree into two and splits the wavefront by introducing a ***splitting edge***. It should be remarked that this ***split event*** is *not the same* as the split event in the straight skeleton; however, it accomplishes roughly the same thing, namely to split the sweeping polygon and shrinking tree into two and recursively continue the sweep simultaneously in each resulting side. As with

the straight skeleton, each wavefront ends when it contracts to a single point called a ***peak***.

The ***universal molecule***, like the straight skeleton, is defined by tracing the vertices of the wavefront during the parallel sweep process (along with certain splitting edges introduced at splitting events). We give significantly more detail on this process in Ch. 5.

CHAPTER 3

PRIOR WORK

3.1 Algorithms for computing the straight skeleton

The straight skeleton, first introduced by Aichholzer et al. in 1995 [4], has generated many research directions in computational geometry over the last two decades. Researchers have looked at a wide-array of problems including: Is there a non-procedural definition for the straight skeleton? If one is given a metric tree T with positive weights at each edge, under what conditions does there exist a polygon P whose straight skeleton is T? Can the straight skeleton be generalized to higher dimensions? What other straight-line skeletons might exist? Etc. One of the main questions addressed in this thesis is the computational one–what is the complexity of computing the straight skeleton? Though work on this problem has spanned twenty years, there is still a significant gap between the fastest algorithms for computing it and the only known lower bounds, which are the trivial $\Omega(n)$ lower bound for polygons, and a lower bound of $\Omega(n \log n)$ for planar straight line graphs (by a reduction to sorting). In this section we survey the growing literature concerning the algorithmic complexity of the straight skeleton, and that of the motorcycle graph, a structure which has been used in the fastest algorithms for computing the straight skeleton for over a decade.

3.1.1 Parallel sweep based based algorithms

We first survey what are, in some sense the most natural algorithms for computing the straight skeleton, namely those that simulate the parallel sweep process to

generate the straight skeleton. One particularly interesting feature of some of these algorithms is that they seem to "work well in practice" even though their theoretical runtimes are worse than the algorithms presented in the next section. They also tend to be the most *implementable*, in the sense that the algorithms are simpler and rely on simpler data structures than those we investigate in the next section. In each of these n denotes the number of vertices in the polygon/PSLG and r denotes the number of reflex vertices.

3.1.1.1 Aichholzer et al., 1995

In the same paper in which the straight skeleton was first introduced for simple polygons [4], Aichholzer et al. suggested computing the straight skeleton by simulating the parallel polygonal sweep directly. Simultaneous events are not dealt with in this work, and thus in the following we assume that no simultaneous events occur. Simulating the sweep requires detecting two types of events–contraction events, which occur when an edge of the parallel sweep contracts to a single point; and splitting events, which occur when a reflex vertex hits some edge elsewhere in the sweep. Naively, this can be simulated in $\Theta(n^3)$ time and $O(n)$ space by employing the following recursive strategy. First compute, for each edge, the time t at which it would contract, ignoring the possibility of splitting events. These are ***candidate*** contraction events. Then for each vertex v and edge edge e not incident to v, compute the time at which v hits e if no other events were to occur before the hit. These are candidate splitting events. Finally, take the minimum candidate event to be the next event, update the state of the sweeping polygon by advancing each edge to the time of the event, and recurse. This algorithm takes $O(n^3)$ time and $O(n)$ space. It is mentioned that this might be improved to $O(n^2 \log n)$ time at the cost of $O(n^2)$ space using a priority queue to store events, though important details, such as how to deal

with the fact that it is possible that at each splitting event $\Theta(n^2)$ events currently in the queue are invalidated, are not mentioned.

3.1.1.2 Aichholzer and Aurenhammer, 1996

In [3], Aichholzer and Aurenhammer generalize the straight skeleton to planar straight-line graphs (PSLGs) and describe an algorithm for computing it. As before this algorithm simulates the parallel sweep process propagating from the edges of the PSLG. The basic idea is to maintain the regions yet to be swept in a kinetic triangulation. As the sweep progresses, a triangle may collapse when its vertices become collinear. These events are categorized into three types. The first correspond to contraction events in the parallel sweep. The second correspond to splitting events in the parallel sweep. The third are ***flip events***. These occur when one vertex of the triangle "hits" an edge shared with another triangle. The two triangles are "flipped" by removing the edge containing the collision vertex and adding an edge from that point to the vertex opposite the removed edge in the opposite triangle. This removes the collision and allows the simulation to proceed.

They observe that since all the points move along linear trajectories, the number of flip events is bounded by the number of times any triplet of the n points become collinear. For a given triplet of points moving along constant linear trajectories, this happens at most twice throughout the motion, which leads to an upper bound of $O(n^3)$ on the number of events processed during the sweep. The triangulation contains $O(n)$ triangles, and the collapse time of each triangle is stored in a priority queue. Processing flip events requires updating $O(1)$ events in the queue, while processing contraction or splitting events may require $O(n)$ changes to the queue. However, since there are $O(n)$ such events in total, this leads to a total of $O(n^2 \log n)$ time spent processing just the contraction or splitting events. The best known analysis for the remaining number of flip events, however, is that there are $k = O(n^3)$ flip

events. Thus the algorithm takes $O((n^2 + k) \log n) = O(n^3 \log n)$ time, which is theoretically worse than the $\theta(n^3)$ naive algorithm (which follows the same approach as in the case of simple polygons in the last section); however, no known examples exist of a family of polygons for which $\Omega(n^2)$ flip events occur and Aichholzer and Aurenhammer observe that on "real-world data" their algorithm seemed to behave more like $O(n \log n)$. It remains an open problem whether a better theoretical upper bound on the number of flip events can be achieved.

3.1.1.3 Huber and Held, 2010

In a series of papers [38, 39, 40] Huber and Held develop and implement a parallel sweep based algorithm for computing the straight skeleton of PSLGs they call Bone. Their algorithm is in the same spirit as Aichholzer and Aurenhammer's, but instead of using a kinetic triangulation of the yet-to-be-swept ***free space*** "outside" the parallel sweep, they make use of the motorcycle graph (defined in Ch. 2). Unlike the previous algorithms in this section, they explicitly deal with degenerate cases in which multiple events occur simultaneously.

They show that the motorcycle graph subdivides the free space into convex polygons. They then simulate the parallel sweep while maintaining the induced subdivision of the free space induced by the motorcycle graph. Recall that one of the main difficulties in computing the parallel sweep previously is the fact that a reflex vertex may hit some edge elsewhere in the polygon. Here, however, because all faces of the induced subdivision are convex, then events occur only between neighboring vertices. In other words, all events are *local*.

The main work now is in tracking the intersection of the parallel sweep with the motorcycle graph. In order to do this Steiner vertices are added to the parallel sweep at the points of intersection between the parallel sweep and the edges of the motorcycle graph. The main work in maintaining the parallel sweep is in maintaining

these Steiner vertices, since they move, or in the language of Huber and Held "surf" along the motorcycle graph edges, several additional events must be tracked. A *start event* occurs when the parallel sweep encounters a vertex of the motorcycle graph, which requires tracking a new Steiner vertex. A *switch event* occurs when the edge of the parallel sweep intersecting a given edge of the motorcycle graph changes. In other words, a vertex of the parallel sweep "passes through" an edge of the motorcycle graph. In this case, the Steiner vertex corresponding to the intersection of the parallel sweep with the motorcycle edge must be switched from one edge of the parallel sweep to the other. A third event, in which two Steiner vertices meet represents the parallel sweep leaving one edge of the motorcycle graph, and the corresponding Steiner vertex is simply removed. In degenerate cases additional events, *multi split events*, which occur when multiple reflex vertices of the sweep simultaneously encounter each other, and *multi start events* which are start events that occur at motorcycle graph vertices representing simultaneous crashes of motorcycles at the same point.

As before, events are stored in a priority queue, and since events only occur between adjacent vertices in the induced subdivision, there are $O(n)$ events in the queue at any time. We have already seen that there are $O(n)$ contraction and splitting events in a parallel sweep of a PSLG, so analyzing the running time requires bounding the number of the new event types. Since the number of start events is given by the number of vertices in the motorcycle graph, there are at most $O(n)$ of these throughout the running of the algorithm. The number of switch events is bounded by the number of times a given vertex of the sweep can "pass through" an edge of the motorcycle graph, which is exactly once. Thus there are at most $O(n^2)$ such events. This leads to an $O(n^2 \log n)$ runtime and there are known examples for which $\Omega(n^2)$ switch events occur; however, Huber and Held give experimental results that suggest that on "real-world data" the number of switch events is often more like $O(n)$ and the algorithm behaves like $O(n \log n)$.

3.1.2 Roof based algorithms

The parallel sweep based algorithms reviewed in the previous section are conceptually simple and have rise to useful implementations, but have not yet proved fruitful in answering one of our central questions–what is the complexity of the straight skeleton problem? Part of the difficulty seems to be the sequential nature of the sweep and the need to simulate events *in the order in which they occur in the parallel sweep.* This difficulty arises from the procedural nature of the parallel sweep definition of the straight skeleton. Due to this problem, many researchers have sought an alternative characterization of the straight skeleton, one that is not procedural and where the objects on which we are computing can be known (or found) at the beginning of the computation. This search for the golden fleece has proved elusive and challenging. The best approaches that have been found thus far (and the approach we employ in Ch. 4) have, in a sense, off-loaded the procedural nature to the motorcycle graph. Once the motorcycle graph is known, then the slab set (see Ch. 2) can be computed quickly, and the work of computing the straight skeleton becomes the work of computing a lower envelope for this slab set. This is one of the main approaches of the algorithms that follow. In these cases we give the time complexity of the algorithm *once the motorcycle graph is known.* In other words, if the algorithm runs in $O(S(n))$ for some function $S(n)$, then computing the straight skeleton require $O(M(n)+S(n))$ time where $M(n)$ denotes the fastest motorcycle graph algorithm.

3.1.2.1 Eppstein and Erickson, 1998

The straight skeleton algorithms of Eppstein and Erickson [33] bridge the gap between the parallel sweep algorithms of the previous section and the straight skeleton roof-based algorithms in the remainder of this section. Their algorithms still simulate the sweep process, but lift the sweep into $\mathbf{R}^3$. Fast data structures for computing ray-triangle intersections help find the next splitting events in the sweep in sub-linear

time, leading to our first sub-quadratic algorithms for computing the straight skeleton. This work is also notable because it is here that Eppstein and Erickson first introduce the motorcycle graph, though it is not used in the algorithm. Rather, Eppstein and Erickson use it to distill what appears to be the central difficulty in computing the straight skeleton–which, as we have seen, is that arbitrarily small perturbations can lead to arbitrarily large changes in the speed at which a vertex moves in the wavefront, which in turn may lead to drastically different straight skeletons.

Two algorithms are described in this work for computing the straight skeleton. The first is slightly slower, but works in the more general case of weighted straight skeletons in which each edge of the parallel sweep process moves at an arbitrary constant speed, rather than at unit speed.

The basic idea of the first algorithm is to lift the parallel sweep process into $\mathbf{R}^3$ by adding time as the third dimension. In other words, the parallel sweep at time t is embedded in the $z = t$ plane (as per the roof definition, see Ch. 2). At the start of the algorithm, we define to sets of objects that are used to compute the time of the next event. The first is a set of (possibly unbounded) triangles in R^3, one for each edge of the polygon (or PSLG), which is its base. The remaining sides of the triangle are given by extending rays along the trajectories of the endpoints of e in the lifted sweep. The triangle is bounded if its two edges intersect at some point above the xy-plane, call this intersection point the triangle's ***upper vertex***. This occurs if and only if the edge e is shrinking in the sweep. Otherwise, the triangle is unbounded. The second is a set of rays, one for each reflex vertex in the polygon. This ray points along the trajectory of the vertex in the sweep.

A contraction event in the parallel sweep is equivalent to the sweep plane reaching the upper vertex of one of the bounded triangles. A splitting event is equivalent to the sweep reaching the intersection of one of the rays and one of the triangles. The next event, then, is the lower (in terms of z-coordinate) of the lowest upper vertex

among all the bounded triangles and the lowest intersection between a ray and any triangle. Two separate data structures are used. The first is a priority queue that stores the bounded triangles by the height of their upper vertex. This handles all of the contraction events in $O(n \log n)$ time. The second is a dynamic range-searching data structure which maintains the lowest intersection point between a set of triangles and a set of rays. Insertion, deletion, and lowest intersection queries take $O(n^{3/5+\epsilon})$ time in this data structure. Since processing each event requires changing a constant number of triangles, this leads to an $O(n^{8/5+\epsilon})$ time and space algorithm.

The second algorithm is a modification of the first and exploits the fact that each face of the (unweighted) straight skeleton roof has slope 1 to use slightly faster range searching data structures. They replace the triangles with a set of slabs similar to those described in Ch. 2 and use slightly better ray-shooting data structures to achieve an $O(n^{1+\epsilon} + n^{8/11+\epsilon} r^{9/11+\epsilon})$ time and space algorithm (where r denotes the number of reflex vertices).

3.1.2.2 Cheng and Vigneron, 2002

Cheng and Vigneron [22, 23] were the first to reduce the computation of the straight skeleton to that of the motorcycle graph by describing an algorithm for computing the straight skeleton of a polygon (with holes) from its induced motorcycle graph *without simulating a parallel sweep*. This is also our first example of an algorithm that assumes the motorcycle graph is given as input.

The algorithm computes the straight skeleton by computing the part of the lower envelop above the motorcycle slab set slabs(P) (see Ch. 2). The algorithm is divide-and-conquer and randomized. It runs in expected $O(n \log^2 n)$ time (though the worst case is $O(n^2 \log n)$) given the motorcycle slab set slabs(P) as input. The basic idea is to recursively subdivide the interior of the polygon into polygonal cells using random internal nodes of the straight skeleton. Once the cells are small enough (in terms of

the number of edges in each cell), the part of the straight skeleton roof lying above the cell is computed by brute force. A key component of the algorithm is an operation for computing the intersection of a plane orthogonal to the x-axis, called a ***vertical plane***, with the straight skeleton roof. This operation works *without computing the entire straight skeleton* in $O(n \log n)$ time. First intersect each slab in slabs(P) with the vertical plane. The intersection of each slab with the vertical plane is a line segment or ray. Let S denote this set of line segments. The intersection of the straight skeleton roof with the vertical plane is given by the lower envelope of these line segments, which can be computed using standard techniques in $O(n \log n)$ time. In a similar way, a face of the straight skeleton roof can be computed independently from the rest in $O(n \log n)$ time.

The divide step works by first picking a random face of the straight skeleton and computing its explicit representation. Then the straight skeleton vertex incident to the face that is closest to the centroid of the straight skeleton is chosen. (The ***centroid*** of a tree is the vertex such that all of its sub-trees have size at most half that of the straight skeleton.) This can be found in $O(n)$ time once the explicit representation of the face is known. The basic idea is that in expectation, the chosen vertex will not be too far from the centroid. Then a vertical plane through the chosen vertex is intersected with the roof. This intersection is then used to subdivide the tree by tracing rainwater paths (Sec. 2.2.1) down from the vertices of the intersection. These subdivide the interior into polygonal cells and the divide step is recursively invoked on each cell. Once the size of each cell reaches a certain threshold (say fewer than 20 vertices), then the part of the straight skeleton above the cell is found by brute force.

3.1.2.3 Huber and Held, 2010

In the same work in which they introduce Bone [38, 39, 40], Huber and Held also outline an approximate algorithm for computing the straight skeleton on graphics

hardware. The algorithm uses the motorcycle graph to construct 3D models of the slabs within some large enough bounding cube and then uses graphics hardware to render a view of the scene from below via standard rasterization techniques. This technique is essentially a grid sampling of the straight skeleton. It is able, for a regular rectangular grid or lattice, to determine which face of the straight skeleton each lattice point resides on; however, the algorithm is not correct in the sense that small features (e.g. those smaller than the grid unit size) cannot in principle be recovered from this method.

3.1.2.4 Biedl et al., 2014

Beidl et al. [9] describe an algorithm for computing positively weighted straight skeletons of monotone polygons. A monotone polygon is one in which there exists some direction in the plane (called the ***direction of monotonicity***) such that all lines parallel to that direction intersect the polygon in at most two points. A monotone polygon can always be decomposed into two monotone polygonal chains–a chain of edges in the plane such that any line parallel to the direction of monotonicity intersects the chain at most once. They show that (a) the parallel sweep of a monotone chain exhibits only contraction events and no splitting events, and (b) the straight skeleton roof of the monotone polygon can be found by the following procedure. First, construct the straight skeleton terrains for the two monotone chains. The straight skeleton roof of the polygon is the part of the lower envelope of these two terrains restricted to the region above the polygon. The intersection of the two terrains is a connected polygonal chain starting at one end of the two monotone chains and ending at the other end. Once the terrains are computed, finding this path amounts to "walking" along the intersection of the two surfaces which is easily done in linear time. Since computing the parallel sweep in this case only requires processing contraction

events, the terrains can be efficiently computed in $O(n \log n)$ time using a priority queue.

3.1.2.5 Cheng et al., 2014

Cheng et al. [24] present an algorithm for computing the straight skeleton of a polygon (possibly with holes) in $O(n \log n \log r)$ time (where r is the number of reflex vertices in the polygon). The basic idea of their algorithm is to subdivide the polygon into cells so that the part of the straight skeleton roof lying above each cell is convex. The algorithm requires two passes, a vertical subdivision algorithm which recursively applies a vertical intersection operation through all vertices of the motorcycle graph to obtain a first subdivision of the polygon. The cells in this subdivision do not yet have the desired property, but the motorcycle edges that intersect the cell form a special outer-planar graph, which means each one passes all the way through the cell (rather than, say, ending on its interior). A second sub-routine further subdivides these cells along the motorcycle edges to obtain the desired subdivision. Along the way the slabs supporting the faces of the straight skeleton that lie above each cell is maintained. Once the final subdivision is obtained, computing the part of the straight skeleton over each cell reduces to computing the lower envelope of a set of planes in $\mathbf{R}^3$, which can be computed efficiently using standard techniques. We use a modified version of the vertical-subdivision step from this paper in Ch. 4.

3.2 Algorithms for computing the motorcycle graph

In the discussion above, we have given an overview of the development of faster algorithms for computing the straight skeleton. Several of these algorithms require the induced motorcycle graph as input. Thus to analyze the running times of these algorithms, we also need the fastest motorcycle graph algorithsm. We give a brief overview of the history of the theoretically fastest motorcycle graph algorithms here.

As we have noted, Eppstein and Erickson [33] were the first to introduce the motorcycle graph as a model of the main difficulty encountered in computing the straight skeleton. They gave an $O(n^{17/11+\epsilon})$ time algorithm for computing, which used essentially the same techniques as their straight skeleton algorithm, but did not show any formal connection between the straight skeleton and the motorcycle graph.

The first to formally use the motorcycle graph in the computation of the straight skeleton were Cheng and Vigneron [22, 23]. They give an $O(n\sqrt{n}\log n)$ time algorithm for computing it using a concept known as a $(1/\sqrt{n})$-cutting. Given a set L of n lines, $(1/\sqrt{n})$-cutting is a subdivision of the plane into cells such that each cell intersects at most $O(\sqrt{n})$ lines from L. Such a cutting with $O(n)$ cells can be constructed in $O(n\sqrt{n})$ time by an algorithm of Chazelle [20]. To compute the motorcycle graph of n motorcycles, Cheng and Vigneron construct an $(1/\sqrt{n})$-cutting of the lines supporting the trajectory of each motorcycle. The part of the motorcycle graph lying on the interior of each cell can then be simulated only for motorcycles traveling on the lines intersecting the cell (of which there are at most $O(\sqrt{n})$).

The fastest motorcycle graph algorithm is that of Vigneron and Yan [58]. There algorithm computes the motorcycle graph of n motorcycles in $O(n^{4/3+\epsilon})$ time.

The algorithms of Cheng and Vigneron [22, 23] and Vigneron and Yan [58] only work in non-degenerate cases. In degenerate cases, Eppstein and Erickson's algorithm [33] remains the fastest.

3.3 State of the art for straight skeleton computation

The following table summarizes the state of the art for straight skeleton computation prior to this thesis. In Ch. 4 we give a new algorithm for computing the straight skeleton given the induced motorcycle graph which allows is an improvement of each of these cases. In the first four cases the fastest algorithms first employ a motorcycle

graph algorithm (column MG) and then a straight skeleton (SS) algorithm. In the last two cases the fastest algorithm remains Eppstein and Erickson's original [33].

	Running time	MG	SS
Non-degenerate polygon	$O(r^{4/3+\epsilon} + n(\log r)\log n)$	[58]	[24]
Degenerate polygon	$O(r^{17/11+\epsilon} + n(\log r)\log n)$	[33]	[24]
Non-degenerate poly. w. holes	$O(r^{4/3+\epsilon} + n(\log r)\log n)$	[58]	[24]
Degenerate poly. w. holes	$O(r^{17/11+\epsilon} + n(\log r)\log n)$	[33]	[24]
Non-degenerate PSLG	$O(n^{8/11+\epsilon}r^{9/11+\epsilon} + n^{1+\epsilon})$		[33]
Degenerate PSLG	$O(n^{8/11+\epsilon}r^{9/11+\epsilon} + n^{1+\epsilon})$		[33]

3.4 Origami and the universal molecule

We are primarily interested in Lang's universal molecule algorithm, a subroutine of Lang's TreeMaker method, which first appeared in [44], and has been represented in several publications [31, 48, 45, 46]. Thought it has appeared in several places, none of the presentations differ significantly from the original. In [48], Lang and Demaine, show how to assign "mountain" and "valley" labels to the crease pattern produced from TreeMaker so that the entire model folds flat in the plane. This is interested, because in general acquiring such a mountain-valley assignment is NP-hard [8]. We now give a brief overview of the growing field of computational origami. Though only twenty years old, the field is growing immensely and is attracting researchers from computer science, mathematics, engineering, physics, material science, and education. To give a complete survey of the field, then, is not our purpose. Instead we provide a brief guided tour. The interested reader is referred to richly illustrated books and conference proceedings for more information [46, 31, 47, 59].

Origami design. Lang's TreeMaker method [44] is a seminal work in the field of computational origami. It is an exemplar of an *origami design problem*, which typically asks how to design a crease pattern for a square or polygonal sheet of paper

so that the paper can be folded along the creases of the pattern to form some desired shape. Lang's TreeMaker is implemented in freely available software [45, 1] and has been used as a tool by origami artists to create many beautiful and complex origami sculptures (cf. [1]). In addition to Lang's work, solutions to other design problems have been studied.

The fold-and-one-cut problem of[28] asks how to fold a sheet of paper with a polygon (or set of polygons, or general PSLG) drawn on its interior so that a single cut from a pair of scissors separates the interior of the polygon from its exterior.

The silhouette folding problem, solved by Demaine et al. [29], asks how to fold a square sheet of paper flat so that its outline is equal to a given simple polygon.

The polyhedron folding problem asks if any polyhedron can be folded out of a single sheet of paper. Demaine et al. present a method which folds a square sheet of paper into a long skinny rectangle and then "wrap" the polyhedron with it [29]; however, their method is highly impractical. More recently Tachi [52] gave a heuristic method called Origamizer that takes as input a desired 3D polyhedral surface and outputs both a polygon (the initial shape of the paper) and a crease pattern that can be folded into the surface.

Both Lang's TreeMaker and Tachi's Origamizer are practical design methods, but both rely on heuristics that sometimes fail to produce an output for a given valid input.

Origami Mathematics. More in the vein of this thesis, other researchers have focused on addressing precise mathematical issues. Huzita developed an axiom system for describing certain origami folds [41]. Ida et al. formalized Huzita's axioms and designed an automated proof assistant that, given an origami construction (e.g. for folding an equilateral triangle out of a square sheet of paper), can automatically prove certain geometrical properties of the construction (e.g. that the folded equilateral triangle has the maximum area possible) [43, 42].

Bern and Hayes showed that it is NP-hard, given a general crease pattern, to decide whether it can be folded flat into the plane [8].

More recently Demaine and Fekete showed that the problem solved by the first phase of TreeMaker is NP-hard [30], and thus the optimization step employed by TreeMaker is a heuristic. On the other hand, the the universal molecule, which is the second phase of TreeMaker and a key ingredient, is computable in polynomial time; however, to our knowledge, before the present work no analysis of the algorithm has appeared nor have any implementation details.

TreeMaker and the universal molecule. One practical difficulty with TreeMaker is that its first phase may result in non-convex polygons, but its second phase, which uses the universal molecule algorithm to fill in each polygon can only work if all of the resulting polygons are convex. In this case it simply fails and the user is required to tweak the input in order to proceed. Because of this, it is of practical importance to extend the universal molecule algorithm to non-convex polygons. Demaine and O'Rourke conjectured that this might be possible (Conjecture 16.8.1 of [31]), but no algorithm proving the conjecture has yet appeared.

A major hindrance in finding such an extension is that although the algorithm has been known now for almost twenty years and has appeared in several publications [44, 31, 45, 48], a precise mathematical formulation of its input, its output, and the connections between the input and output independent of the algorithm had not ever appeared. In fact, the most comprehensive treatment of the universal molecule is in Demaine and O'Rourke's book [31], in which they mention that a full proof of correctness would require a careful treatment which they do not attempt.

Rigid foldability. One further question arising from computational origami is whether a particular origami crease pattern can be rigidly folded as a panel-and-hinge structure to some final state. This is important for certain applications, for

instance in the development of deployable structures like satellite arrays. The most general case of rigidly foldable origami which is settled is that of single vertex origami [50, 49] which can be folded rigidly in a non self-intersecting manner.

Tachi has shown how to produce several specific origami structures that are rigidly foldable (cf. [54, 53]) and has described an optimization algorithm that modifies an input origami crease pattern so that it is approximately rigidly foldable, and simulates the kinematics to produce an approximate folding animation [51].

For the universal molecule crease patterns the story is only partially known. Demaine and Demaine [27] showed that if the universal molecule crease pattern for a polygon is precisely its straight skeleton, then the crease pattern is rigidly foldable . Their proof is constructive and shows how to determine the motion from the starting flat state to the final folded state. However, if we fix the input metric tree, then the set of Lang polygons compatible with the tree for which the universal molecule is the straight skeleton forms a measure zero set in the space of all possible compatible Lang polygons.

CHAPTER 4

COMPUTING THE STRAIGHT SKELETON

In this chapter, we show how to compute the straight skeleton of a polygon with n vertices from its induced motorcycle graph in $O(n \log n)$ time and for a planar straight line graph with m connected components in $O(n(\log m) \log n)$ time.

Instead of computing the straight skeleton directly, our algorithm computes the straight skeleton roof (first defined in Ch. 2) by finding the lower envelope of a set of partially infinite slabs in $\mathbf{R}^3$. The main bulk of the work is to describe the algorithm for polygons. We then extend our algorithm to planar straight line graphs (PSLGs) using a modification of the recent cellular subdivision algorithm of Cheng, Mancel, and Vigneron. We use the subdivision algorithm to divide the plane into regions which can then be "filled in" by our polygon-based algorithm.

This work is currently under review and a preprint is available on the ArXiV [12, 11].

4.1 Introduction and Background

Recall from Sec. 2.2 that the straight skeleton is a particular tree drawn on the interior of a polygon. It is often defined by a wavefront process in which each edge of the polygon moves inwards in parallel at unit speed (Fig. 4.1). The trace of the vertices of the wavefront forms the straight skeleton. The wavefront is extended to planar straight line graphs (PSLGs) [3] by generating a square cap around each degree 1 vertex (Fig. 4.2). The straight skeleton of a PSLG is defined similarly as the trace of the vertices during the wavefront. Though its cousin the medial axis can be computed

	Previous work	This work
Non-degen. polygon	$O(r^{4/3+\epsilon} + n(\log r)\log n)$	$O(r^{4/3+\epsilon} + n\log n)$
Degen. polygon	$O(r^{17/11+\epsilon} + n(\log r)\log n)$	$O(r^{17/11+\epsilon} + n\log n)$
Non-degen. poly. w. holes	$O(r^{4/3+\epsilon} + n(\log r)\log n)$	$O(r^{4/3+\epsilon} + n(\log h)\log n)$
Degen. poly. w. holes	$O(r^{17/11+\epsilon} + n(\log r)\log n)$	$O(r^{17/11+\epsilon} + n(\log h)\log n)$
Non-degen. PSLG	$O(n^{8/11+\epsilon}r^{9/11+\epsilon} + n^{1+\epsilon})$	$O(r^{4/3+\epsilon} + n(\log m)\log n)$
Degen. PSLG	$O(n^{8/11+\epsilon}r^{9/11+\epsilon} + n^{1+\epsilon})$	$O(r^{17/11+\epsilon} + n(\log m)\log n)$

Table 4.1: Comparison of straight skeleton algorithms for the polygon, polygon with h holes, and PSLG with m connected components. In each case n denote the number of vertices and r denotes the number of reflex vertices.

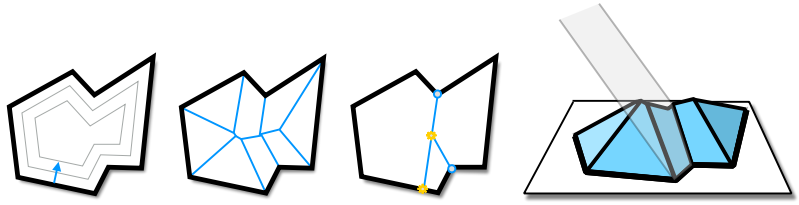


Figure 4.1: The wavefront, straight skeleton, induced motorcycle graph, and straight skeleton roof for a polygon. The gray shaded strip is one of the slabs in the polygon's slab set.

efficiently, algorithms for the straight skeleton are still significantly slower than the known lower bounds. The current state of the art and a comparison with the present work is summarized in Table 4.1.

We briefly review the relevant literature here. Chapter 3 contains a more detailed literature review. In the case of a polygon (possibly with holes), the fastest algorithms work by computing a secondary structure called the ***induced motorcycle graph*** and then use the induced motorcycle graph to compute the straight skeleton. This is the approach taken in this chapter. We operate in the ***roof/terrain model***, which defines the straight skeleton as a polyhedral surface in $\mathbf{R}^3$. Eppstein and Erickson showed that this surface is equivalent to the lower envelope of a set of partially infinite strips in $\mathbf{R}^3$ called slabs. Cheng and Vigneron [23] gave a slightly different set of slabs defined with respect to the motorcycle graph with the same property which was later extended to degenerate cases by Huber and Held [39]. Our algorithm computes this lower envelope using divide and conquer on the set of slabs from [23, 39]. One novelty

of our approach is that while the final straight skeleton roof is indeed a terrain, the intermediate sub-structures computed by our algorithm, which we call ***partial roofs***, are neither terrains nor lower envelopes. Our algorithm is deceptively simple to state: we divide the polygon into two subchains, recursively compute partial roofs for the subchains, and then merge the two by computing a *local* walk of the intersection between the two and cutting away the parts of each face "above" the walk before gluing the two surfaces together along the path.

Though conceptually simple, tracking the details of why the algorithm works is involved. We make heavy use of the distinction between the intrinsic properties of a surface and the extrinsic properties of their realization. Topologically, our partial roofs are disks but their realizations may not be in much the same way that a Klein bottle is topologically a manifold but no realization of a Klein bottle in $\mathbf{R}^3$ is a manifold.

The motorcycle graph for r motorcycles can be computed in $O(r^{4/3+\epsilon})$ time in non-degenerate cases using the algorithm of Vigneron and Yan [58] or $O(r^{17/11+\epsilon})$ time in degenerate cases using the algorithm of Eppstein and Erickson [33] and the extended definitions from [39]. In addition to this work there are three other known algorithms that use the motorcycle graph as input to compute the straight skeleton. The first is the randomized $O^*(n\sqrt{h+1}\log^2 n)$ time algorithm of Cheng and Vigneron [23] which works for a polygon with h holes. The second is the $O(n^2 \log n)$ time kinetic simulation based approach of Huber and Held [39] which works in the general case of PSLGs. One intriguing aspect of that work is that though the algorithm takes $O(n^2 \log n)$ time, it is shown experimentally that on most inputs it behaves more like $O(n \log n)$. The third, which was found simultaneously with the polygon algorithm we present here, is due to Cheng et. al and takes $O(n(\log r)\log n)$ time for a polygon (possibly with holes) with r reflex vertices [24]. We use a modified version of the cellular

subdivision algorithm from that work–except with a smaller depth of recursion–to extend our polygon algorithm to PSLGs. The main result of this chapter is:

Theorem 4.1.1. *Given the induced motorcycle graph as input, the straight skeleton of an n-vertex polygon can be computed in $O(n \log n)$ time and an n-vertex PSLG with m connected components can be computed in $O(n(\log m) \log n)$ time.*

We note that in the case of PSLGs, the fastest algorithm for computing the straight skeleton did not use the motorcycle graph as input, since the only known reduction from the straight skeleton to the motorcycle graph required $O(n^2 \log n)$ time. Our algorithm removes this bottleneck and brings the PSLGs in line with the polygon cases.

4.2 Preliminaries

We now recall some of the definitions from Ch. 2 that are used in this chapter.

Straight skeleton roof. The following properties of the straight skeleton are needed in this chapter. The first applies only to the straight skeleton of a polygon and the second applies to both polygons and PSLGs.

1. The straight skeleton of a polygon is a tree and its leaf nodes are the vertices of P.

2. The straight skeleton subdivides the polygon (or PSLG) into faces. Each face is incident along a single ***base edge*** of P and is monotone with respect to its base.

The ***straight skeleton roof*** of a polygon P is defined by lifting each vertex into $\mathbf{R}^3$ by setting its z-coordinate to the time t at which it appeared in the wavefront (cf. [4]). Each face lies in the plane Π_e through its ***base (polygon) edge*** e that makes an angle of $\pi/4$ with the interior of the polygon. Each face is monotone in Π_e with

respect to its base edge. The roof, denoted R_P, is a polyhedral terrain in $\mathbf{R}^3$ that is topologically a disk. The definition sketched above refers to the wavefront; however, we use a different characterization as the lower envelope of a set of partially infinite strips called ***slabs*** in $\mathbf{R}^3$ which we define shortly.

Motorcycle Graphs. Before defining the slab set we use, we first review the definition of the motorcycle graph. Recall from Ch. 2 that a motorcycle graph is given by first placing a number of "motorcycles" $m_1, \ldots, m_n$ in the plane. Each motorcycle moves along a linear trajectory at constant speed. As a motorcycle moves it leaves a track behind it. If one motorcycle encounters the track of another it crashes. Each motorcycle either crashes or escapes, in the sense that it reaches a point at which it is moving along its supporting line away from the intersection of its supporting line with the supporting lines of all other motorcycles. The motorcycle graph is given by the trace of all motorcycles throughout this process. We say that a motorcycle graph is ***generic*** if no two motorcycles crash into each other simultaneously.

Induced motorcycle graph of a polygon. Given a polygon P, its ***induced motorcycle graph*** $\texttt{MG}_P$ is defined by placing a motorcycle at each reflex vertex v with a velocity of magnitude $1/sin(\theta_v/2)$ (where θ_v is the interior angle at v) and direction pointed inwards along the angle bisector at v. The motorcycles then move as before while laying down tracks. The only difference is that now both the polygon's edges and the tracks are obstacles that the motorcycles crash into. We say that the induced motorcycle graph is ***generic*** if no two motorcycles simultaneously crash into each other. Our definition thus far suffices only for generic cases.

Extending to non-generic cases. To extend to non-generic cases we use the construction from [39]. Under certain conditions, when multiple motorcycles simultaneously crash into one another, we will remove the crashed motorcycles but "spawn" a new motorcycle with a prescribed direction. We will call this a ***multi-crash event***

and a "normal" crash event a ***solo-crash event***. Before defining exactly how this is done, we need to track a little more information at each motorcycle. Now, for each motorcycle we define a ***left arm*** and a ***right arm***. These are vectors with origin at the moving motorcycle and direction we will define shortly. We first define the left and right arms for each initial motorcycle m. Remember that the initial position of m is at a reflex vertex v of the polygon and thus we have a well-defined left and right edge (when looking towards the polygon's interior from the vertex). We define the direction of the left (resp. right) arm vector to be pointing along the left (resp. right) edge of v outwards from v. At certain multi-split events we will spawn a new motorcycle, and will therefore need to define its left and right arms.

Handling multi-crash events. Let $m_1, \ldots, m_k$ denote all k motorcycles that crash simultaneously into each other at a multi-crash event. Let p denote the point at which the event occurs. Without loss of generality, assume that m_1 through m_k are indexed by the counter-clockwise order of the tracks left by m_1 through m_k at p. The traces of the motorcycles subdivide the plane at p into k "wedges". Either all of these wedges are convex, in which case the motorcycles m_1 through m_k simply crash and we do not spawn a new motorcycle, or one wedge–without loss of generality assume the counter-clockwise wedge from the track of m_k back to the track of m_1–is non-convex. In this case we spawn a new motorcycle m' according to the following rules.

1. Let L denote the left arm of m_1 and R denote the right arm of m_k. If the counter clockwise angle from R to L is reflex, then we spawn m' along the bisector of this angle. The left arm of m' is L and the right arm of m' is R.

2. Otherwise, if the angle is convex, then m' continues the motion of m_k and inherits its left and right arms from m_k.

Motorcycle arm chains. Further, we define two ***motorcycle arm chains*** for each reflex vertex v in the polygon. One chain is defined for the left edge incident to v and the other is defined for the right edge, which we call the ***left*** and ***right motorcycle arm chains*** for v. Let m be the initial motorcycle spawned at v.

The left motorcycle arm chain. The left arm chain is defined inductively by first tracing m. If m crashes in a solo-crash event, then we are done, and the left motorcycle arm chain of v ends at the crash point. If m crashes in a multi-crash event between motorcycles $m_1, \ldots, m_k$, then whether we continue depends on whether m is m_1 or not. If m is in $m_2, \ldots, m_k$ then the left motorcycle arm chain ends at the crash site. On the other hand, if $m = m_1$, then it depends on whether we applied rule 1 or rule 2 to spawn the new motorcycle m'. In the case of rule 1, then we continue adding to the left motorcycle arm chain by tracing m'. If, however, we proceed by rule 2, then the left motorcycle arm ends at the intersection point.

The right motorcycle arm chain. The right arm chain is defined analogously with a slight change at multi-crash events. If m crashes at a multi-crash event, and m is not m_k, then it simply crashes. If, however, $m = m_k$, then regardless of whether we apply rule 1 or rule 2, the trace of the right motorcycle arm chain continues by tracing the motion of m'.

For a PSLG we launch a motorcycle out from any reflex angle made in the PSLG. We launch two motorcycles from any 1-degree ***terminal vertex*** such that they make angles of $3\pi/4$ and $5\pi/4$ with the edge incident the terminal vertex. See Fig. 4.2.

Induced motorcycle graph of a PSLG. Huber and Held [39] extended the induced motorcycle graph to PSLGs. In addition to launching motorcycles from each reflex vertex, we launch two motorcycles from each vertex v of degree 1, one making an angle of $3\pi/4$ and the other making an angle of $5\pi/4$ with the edge incident to v. The properties of the induced motorcycle graph needed for this chapter (which were proved in [37]) are:

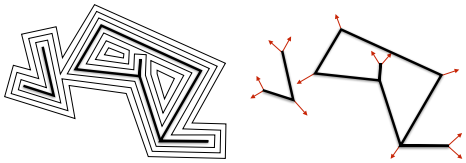


Figure 4.2: The wavefront and motorcycles for a PSLG. Terminal vertices generate a square cap which necessitates sending out two motorcycles.

1. The motorcycle arm chains of a reflex vertex v are convex. Furthermore, the left motorcycle arm chain of v is monotone with respect to the left edge of v and the right motorcycle arm chain is monotone with respect to the right edge of v.

2. In generic cases each motorcycle arm chain has a single edge, and in non-generic cases there are a total of $O(n)$ edges over all motorcycle arm chains.

Slabs and the lower envelope. Following Huber and Held [39] we define a single slab for each edge e of the polygon. The slab lies in the plane Π_e through e that makes an angle of $\pi/4$ with the interior of P. For each reflex vertex v of e if e is to the right (resp. left) of v, lift its right (resp. left) motorcycle arm chain upwards onto Π_e. The ***slab*** s_e for e is defined as the region of Π_e above (with respect to the slope vector of Π_e) the edge and (lifted) motorcycle arm chains for e (we call this the ***lower convex chain***[1] of e). See Fig. 4.3. Denote the slab set for P by S_P. The following gives the characterization of the straight skeleton that we use in the remainder of this chapter. It was proved by Cheng and Vigneron [23] for non-degenerate polygons with holes and was extended to degenerate cases and PSLGs by Huber and Held [39]:

Theorem 4.2.1 ([23, 39])**.** *The lower envelope of S_P (resp. S_G) is the straight skeleton roof of the polygon P (resp. PSLG G).*

We also note the following alternative useful characterization of a single face f with base edge e of the roof R_P. Intersect all other slabs in S_P with s_e resulting in a

[1]By lower convex chain we mean a convex chain where all of the convex angles open upwards.

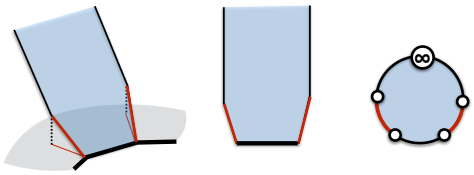


Figure 4.3: *Left:* a slab defined by lifting the motorcycle arms (red) of its base edge up onto the plane through the base edge making an angle of $\pi/4$ with the interior of the polygon. *Center:* the *local view* of the slab. *Right:* the combinatorial view as a closed polygon with one vertex at infinity.

set of line segments on s_e. The lower envelope of the line segments (in the plane Π_e) is equal to f. We call this the ***local lower envelope***.

Properties of the face supported by a slab We now summarize the main properties of a face f of R_P with base edge e:

1. f is monotone w.r.t. e in s_e.

2. f is bounded below by a subset of the lower convex chain of s_e that includes all of the base edge e and a portion of the motorcycle arms of e.

3. f is bounded above by an upper monotone chain of edges, each of which is interior in R_P and lies on the intersection of s_e with another slab in S_P.

Non-generic input. A straight skeleton of a polygon (or PSLG) is ***generic*** if its induced motorcycle graph is generic and if no four slabs intersect at the same point. Our algorithm works in all cases where the lower envelope definition holds, including non-generic situations in which more than three slabs intersect at a point. In a PSLG an edge may be incident to the same face on both sides. We view a PSLG as a *half-edge* or *dart* structure [35], where each edge is split into two twin half-edges in either direction. We consider the two half-edges to be infinitesimally close but not touching. At a terminal vertex we add a zero-length dummy edge, which we consider to make a right angle with both its incident edges, which is the generator of the square cap in the wavefront around a terminal vertex (See Fig. 4.2).

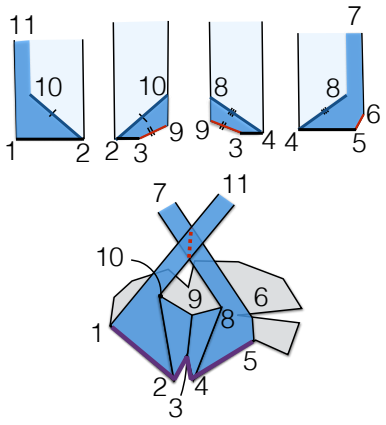


Figure 4.4: The intrinsic (left) and extrinsic (right) view of a partial roof. The vertices are numbered (including vertices 7 and 11 at infinity) to make plane how faces are glued together. Note that the intrinsic surface has no edge between the leftmost and rightmost faces, but in the extrinsic realization the two intersect (dotted red line). Intrinsically, the surface is a topological disk, even though if we forget the underlying combinatorics and only look at the extrinsic realization, it has a self-intersection.

4.3 Partial Roofs

Given a polygon P, we compute its roof R_P using divide and conquer. One might imagine dividing the polygon into two subchains C_1 and C_2, recursively computing the lower envelopes of the slab sets S_{C_1} and S_{C_2}, and then merging the result to obtain R_P. Such an approach is correct, but is too slow for our purposes–for one, the lower envelope of slabs for a chain with n edges may have combinatorial complexity $\Omega(n^2\alpha(n))^2$ [32]. In the following we use this same basic approach, but instead of computing the lower envelope of the slab set of a subchain, we compute a new structure we call a partial roof as the intermediate sub-problem.

A partial roof is a polyhedral surface (henceforward, ***surface***) defined for a subchain C of the polygon that satisfies four properties, to be defined shortly–*face construction*, *face containment*, *edge existence*, and *edge containment*. Like the straight skeleton roof, a partial roof has a single face on each slab in S_C. Before defining precisely the object in question, let us highlight two of its novelties:

- Unlike the faces of R_P, a partial roof can have unbounded faces; however, these faces may intersect without there being an edge between them. This

[2]Where $\alpha(n)$ denotes the inverse Ackermann function.

underscores an important novelty: we make use of the distinction between a surface's underlying ***intrinsic geometry*** and the ***extrinsic geometry*** of the realization it happens to be in. The way to think of a partial roof is this–each face is "cut out" of its slab, completely independently of the others. Faces are then "glued" together by identifying edges (resulting in ***internal edges***). Each internal edge must lie on the intersection of the two slabs supporting its incident faces, but *not all intersections of faces will correspond to an edge.* An example is shown in Figure 4.4. The reader may find it helpful to think of the familiar Klein bottle, which is intrinsically a manifold but any realization of a Klein bottle into $\mathbf{R}^3$ exhibits self-intersections. A flatlander (as it were) walking along a Klein bottle may pass through a self-intersection point but would have no way of detecting it (call this a ***local walk***).

- That said, we will ensure that certain edges along the intersections of slabs *do exist* in their corresponding faces. Namely, if an edge exists in the final roof R_P between the faces for slabs s_1 and s_2 and $s_1, s_2 \in S_C$, then there *must exist* an edge between the faces of any partial roof for C between s_1 and s_2. We will call such an edge a ***critical edge***. See the *edge existence* property below.

  Definition 4.3.1. *Let C be a sub-chain of a polygon P and e be an edge in the final roof R_P for P. Let s_1 and s_2 denote the slabs supporting the two faces incident on e in R_P. We say that e is a* ***critical edge*** *if $s_1 \in S_C$ and $s_2 \in S_C$.*

The purpose of all this is to get around the fact that computing the lower envelope is too slow. Our merge operation ignores or discards intersections that are provably not part of the lower envelope, and focuses only on computing intersections that are or might be.

Definition of a partial roof for a subchain C. A ***partial roof*** R for a subchain C of a polygon P is a (polyhedral) surface, topologically a disk (though its realization

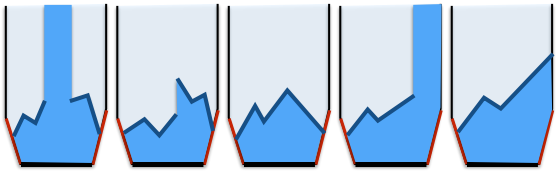


Figure 4.5: A cataloguing of possible face types drawn on their respective slabs. Those with two upper monotone chains both unbounded and bounded, and those with one chain that touches both motorcycle arms, one that stops on the interior, and one that stops on the slab boundary.

may not be) with C as a connected subchain along its boundary. The partial roof is any such surface that satisfies the following properties:

- ***Face construction***. Each slab $s \in S_C$ supports a single face f_s in R. The face may be either bounded or unbounded, and is defined by a subchain of the slab's lower convex chain containing the entire base edge (we abuse the terminology and call this the ***lower convex chain*** of f) and zero, one, or two ***upper monotone chains***. Each upper monotone chain has one endpoint (its ***starting point***) along one of the motorcycle arm chains of the lower convex chain (or the endpoint of the base edge). If there is only one upper monotone chain, then its other endpoint may either be on the interior of the slab or along the opposite side of the lower convex chain. If there are two, then the two chains are ***co-monotone***, meaning that no orthogonal line through the base edge passes through the interior of both chains. The face f_e is defined as the region above (with respect to the direction of monotonicity) the lower convex chain and below the upper monotone chain(s). A face may be unbounded, but combinatorially we connect the two endpoints of its monotone chains by two ***unbounded edges*** incident to a ***vertex at infinity*** as we did with the slabs (See Fig. 4.3). Figure 4.5 shows a zoo of possible faces. Alternatively, if the endpoint of one monotone chain lies monotonically above the endpoint of the other, then we "close" the face by adding a ***monotone edge*** between the two which is on the boundary of R (see the second face in Fig. 4.5). Each monotone chain edge is ***internal*** in R and lies on the intersection of the two

slabs supporting its two incident faces. Motorcycle edges of a face may be internal or boundary edges in R. The base edges and unbounded edges are always boundary edges in R.

- ***Face containment***. Let f_s denote the face supported by a slab s in the partial roof and f'_s denote the face of the final straight skeleton roof supported by s. Then (geometrically) $f'_s \subseteq f_s$. In other words, each face of a partial roof geometrically contains its corresponding straight skeleton face.

- ***Edge existence and containment***. Suppose the straight skeleton roof has an edge e' on the intersection of slabs s_1 and s_2 such that both base edges of the slabs are in C. Then there must exist an edge e incident to the faces f_1 and f_2 supported by s_1 and s_2 in any partial roof for C (***edge existence***) and (geometrically) $e' \subseteq e$ (***edge containment***). We call e and e' corresponding ***critical edges***. In other words, any intersection between slabs in S_C supporting an edge in the final straight skeleton roof also supports an (equal or larger) edge in any partial roof.

Implications Let us examine several consequences of this definition. Lemma 4.3.2 shows that the only partial roof for the entire polygon P is the final straight skeleton roof itself. Lemma 4.3.4 shows that a partial roof has linear complexity in the number of edges in its slab set. In the next section we show how to merge partial roofs for coincident subchains of a polygon to get a partial roof for the entire chain. Lemma 4.3.5 is a straightforward property of the critical edges incident to a single face of a partial roof that nevertheless plays an important role in the proofs of the next section.

Lemma 4.3.2. *The straight skeleton roof R_P is* the *partial roof for P.*

Proof. We first remark that the straight skeleton roof satisfies all of the properties of the partial roof, which is verified by straightforward definition checking. We now

show that it is the only partial roof. Let R be any partial roof for P, s be a slab in S_P, and f_s be the face supported by s in R and let f'_s be the corresponding face supported by s in the straight skeleton roof R_P for P. Since our chain is the entire polygon P, all edges of R_P are critical edges. By the edge existence and containment properties, for each edge e' in f'_s there is a corresponding edge along the same intersection in f_s that geometrically contains it. We next show that each edge e of f_s is equal to its corresponding critical edge e'. Suppose not, then there exists an edge e in f_s that is longer than its corresponding critical edge e' (and contains it). Thus the boundary of f_s geometrically contains the boundary of f'_s. The part of e not contained in e', then, is a segment on the boundary of f_s that makes f_s either non-simple, or non-monotone, contradicting the face construction property. A similar contradiction is reached if we assume any extra edge exists that does not correspond to a critical edge. Thus the edges of f_s are the same as the edges of f'_s. □

Observation 4.3.3. *Each slab is a partial roof for its base edge.*

The next lemma is straightforward due to the fact that the roof is a disk, but we include a proof in the appendix for completeness:

Lemma 4.3.4. *Any partial roof for a subchain C of a polygon such that the slab set S_C has k base and motorcycle edges has $O(k)$ vertices, edges, and faces.*

Proof. The only boundary edges are the base edges, possibly some motorcycle arm edges, and the unbounded edges in unbounded faces. Thus the number of boundary edges and vertices is $O(k)$. Since there are no internal faces, the internal edges form a forest with all leaves on the boundary and no internal vertex has degree 2. Thus each face is incident to at most 4 leaves of the forest, so the total number of leaves is $O(k)$, and thus the total number of internal vertices and edges is $O(k)$. □

Finally, we show that each critical edge incident to a face f of R is part of a path of critical edges that begins at the base edge. This lemma is important in the proof of

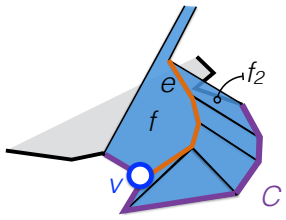


Figure 4.6: An illustration of Lemma 4.3.5. The chain of edges (orange) from e back to the vertex v (blue) must all be critical.

correctness of the merge operation. It follows from the fact that the straight skeleton is a tree.

Lemma 4.3.5 (The critical edges form a path in a face back to the base edge)**.** *Let e be a critical edge incident to a face f of a partial roof R for a subchain C. Let f_2 be the other face incident to e in R and let v be the vertex of the base edge of f incident to the subchain of C between the base edges of f and f_2. Then there is a chain of critical edges between e and v (Fig. 4.6).*

Proof. There must be a path from e back to v in f of internal edges. Otherwise, f is incident to the boundary between e and v which implies that R is not a disk. Let e' be the critical edge corresponding to e in the final straight skeleton roof R_P and let f' be the face corresponding to f in R_P. Let e'' be any edge between e' and the vertex v in f'. We claim e'' is a critical edge. To see why look at the base edge of the other face incident to e''. This base edge necessarily lies on the chain C, and hence e'' is an intersection between two of the slabs for C making it a critical edge. Thus by the edge existence property there is some edge incident to f that corresponds to e''. Thus for all the edges on the path from e' to v in f', there exists a corresponding edge in f. Finally, by edge containment and face construction, these must form a chain in f (following essentially the same argument as in Lemma 4.3.2). □

4.4 Merging partial roofs

Overview Our goal is to merge partial roofs for subchains that are coincident at a *gluing vertex* into a partial roof for the combined subchains. To do this we first

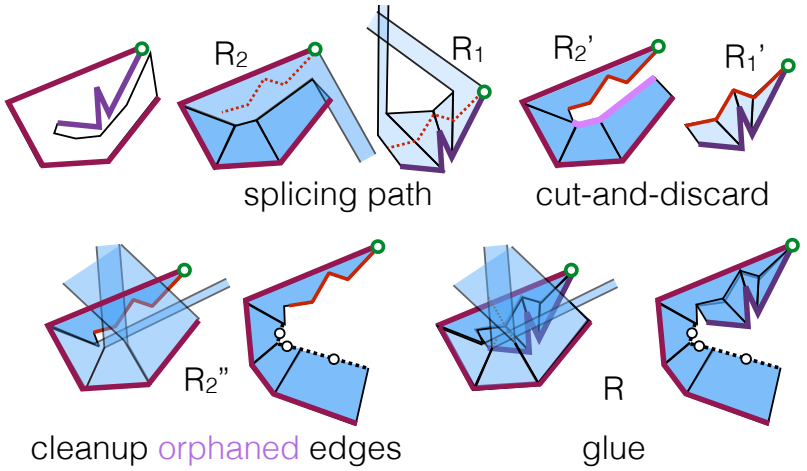


Figure 4.7: The merge operation for partial roofs R_1 and R_2 for the (bold) purple and maroon subchains incident at a gluing vertex. After cut-and-discard R_2' has three orphaned edges (lilac) which are removed by the cleanup operation. In the cleanup operation we show the realization of R_2'' (left) as well as a combinatorial representation of it (right) with the black circles with white fill representing infinity. Similarly for R after the gluing step.

cut each input partial roof along a path called the *splicing path*. This subdivides each face traversed by cutting along the path. We discard all faces to the right of the splicing path in one partial roof and to the left in the other. We show that the result of this cut-and-discard step is that we are left with exactly two topological disks (Lemma 4.4.7), one from each partial roof that both contain their entire base chains. The splicing path is now on the boundary of each, and we glue the two surfaces together along the splicing path. The purpose of the splicing path is to pick up all of the critical edges needed to satisfy the edge existence property. These form a path in the straight skeleton, and we "pick up" the path by computing a local intersection between the two surfaces. The difficulty is in finding a path that is long enough to contain all the edges we need but is short enough that we can maintain some nice properties (like face monotonicity) that allow us to compute the path efficiently. Before we look at more detail, let us make a few observations and discuss the general ideas.

First, each input face by definition satisfies face containment. Our merge operation essentially cuts and discards parts of faces from each input roof. Our goal then, is to ensure we do not cut a face down so far that it violates the face containment property. Second, any edge needed to satisfy edge containment in one of the input roofs is also needed to satisfy edge containment in the output roof. (Recall that such an edge is needed between two faces only if there exists an edge between the corresponding two faces in the final straight skeleton roof–this property is not changed by merging.) Thus our merge operation needs to keep these edges.

In addition to the edges we need to keep, there are also some edges we need to *find*–the critical edges. It turns out that these form a path in the straight skeleton beginning at the gluing vertex (Lem. 4.4.1) and this ***critical path*** is the first part of the *local intersection path* between the input roofs (Lemma 4.4.2). Instead of using the entire local intersection path, we point out some simple properties that are true of each edge of the critical path (Lem. 4.4.1). We use these properties as a set of necessary conditions for an edge of the local intersection path to be an edge of the critical path. We then compute what we call the ***splicing path***, which is the longest subchain of the local intersection path starting at the gluing vertex satisfying the necessary conditions. The reason for using the splicing path, rather than the entire local intersection path, is that the splicing path has some nice features, particularly monotonicity, that allow us to compute it in linear time (see Sec. 4.4.3) and guarantee that each roof is a disk.

Before we describe the operation, we first investigate the properties of the critical path, the local intersection path, and define the splicing path. In the remainder of this section let R_1 and R_2 denote the input roofs for subchains C_1 and C_2 to the left and right of a ***gluing vertex*** $\hat{v}$. Let R_P denote the final straight skeleton roof for P and let $C = C_1 \oplus_{\hat{v}} C_2$.

4.4.1 The critical path, the local intersection path, and the splicing path

The critical path We are interested in the *critical edges* (Def. 4.3.1), or those edges that need to exist in any partial roof of C to satisfy the edge existence property. Recall that the edge existence property applies only to edges e of the R_P such that the two faces f_1 and f_2 incident to e both have their base edges in C. These can be categorized into three types based on which subchain, C_1 or C_2, contains the base edges of f_1 and f_2. Either both are in C_1, both are in C_2, or one is in C_1 and one is in C_2. In the first two cases there already exists an edge in R_1 (resp. R_2) satisfying edge existence for e (by the edge existence property on R_1 resp. R_2). The remaining edges are between a face with base in C_1 and a face with base in C_2. The following properties are a direct consequence of the fact that the straight-skeleton is a tree, the roof R_P is a disk, and the straight skeleton subdivides the polygon into monotone polygons. Without loss of generality assume that C_1 is before and C_2 is after $\hat{v}$ in the ccw ordering on the polygon P.

Lemma 4.4.1. *The critical edges having one slab in S_{C_1} and one slab in S_{C_2} form a path in the straight skeleton roof R_P that starts at $\hat{v}$. This* ***critical path*** *satisfies the following properties. Let cp_f denote the set of edges of the critical path incident to a particular face f of the straight skeleton roof.*

1. *cp_f is connected. In particular this means that the path visits a face at most once (where a "visit" may be along a connected chain of edges).*

2. *cp_f can be further decomposed into at most three distinct subchains: a chain of valley edges on one of the motorcycle arms of f followed by a chain of ridge edges along the upper monotone chain of f followed by a chain of valley edges along the other motorcycle arm of f.*

3. *The path separates the faces with base edges in C_1 to its left, and the faces with base edges in C_2 to its right.*

Proof. We first show that the critical edges having one supporting slab in S_{C_1} and one in S_{C_2} form a path and then prove properties 1–3.

The critical edges supported by a slab in S_{C_1} and a slab in S_{C_2} form a path. We first show that the edges are connected. Let e denote a critical edge having one supporting slab in S_{C_1} and one in S_{C_2}. Let f_1 bet the face supported by the slab in S_{C_1} incident to e and f_2 denote the face supported by the slab in S_{C_2}. B definition the base edge of f_1 is in C_1 and the base edge of f_2 is in C_2.

Let C' denote the chain of base edges from the base edge of f_1 to the base edge of f_2 that contains $\hat{v}$. By the connectedness of C_1 and C_2 at $\hat{v}$, C' is given by the sub-chain of C_1 from the base of f_1 to $\hat{v}$ followed by the sub-chain of C_2 given from $\hat{v}$ to the base of f_2. Taken together f_1, f_2, and C' bound a disk of faces of the straight skeleton roof. Denote this disk by D. Each face in the disk has its base edge in C'.

We now claim that there exists a path of internal edges back from e to $\hat{v}$. In the case of polygons, this follows directly from the fact that the straight skeleton is a tree. Here, however, we give an alternative proof that only assumes the straight skeleton has no internal faces (but may be a forest). *We do this so that the proof immediately generalizes the straight skeleton of the polygonal cells we use in Sec. 4.5 to compute the straight skeleton of a PSLG.* Assume that no such path exists. Then between e and $\hat{v}$ there must be a face f such that its removal subdivides the disk D into two disks, one containing $\hat{v}$ and one containing e. But f can have only one base edge and therefore only one edge in C'. Thus, C' is disconnected, a contradiction. Therefore a path of internal edges must exist. Now, it follows immediately that all faces to one side of the path have their base edges in C_1 and all faces to the other side of the path have their base edges in C_2 and the claim follows.

Properties 1–3. We now prove the 3 properties in order. These follow almost trivially from the fact that the straight skeleton edges are a tree.

1. Suppose not, then there is some edge e of f from which the critical path leaves f and some edge e' at which it re-enters such that the chain of edges between e and e' of f are internal. Let $e_1, \ldots, e_k$ denote this chain, and let p denote the edges of the critical path that connect e to e'. Since all of the edges of the critical path are internal edges in the straight skeleton, the cycle given by the edges of p and the edges of $e_1, \ldots, e_k$ necessarily bounds some disk of faces which are internal to the straight skeleton. But this contradicts that each face of the straight skeleton is incident to a base edge on the boundary.

2. This property follows from the fact that each face of the straight skeleton is given by a base edge, two motorcycle arm chains, and a chain of ridge edges. The property then follows from property 1.

3. Suppose not. The first edge e of the critical path is the one incident to the gluing vertex $\hat{v}$ connecting C_1 to C_2, which has the property by definition. Let e' be the first edge where this is not true. In other words, at e' the face with a base edge e_1 in C_1 is to the right and the face with a base edge e_2 in C_2 is to the left. Let f' be the face incident to both C_2 and e'. Let p_1 and p_2 denote the two paths of base edges that start from the base of C_2 and traverse base edges back to the gluing vertex $\hat{v}$. One of these necessarily contains the e_1 and the other necessarily contains the base edge of the face opposite f' at e', which has its base in C_1 (since the straight skeleton is a disk). But this implies that C_2 is disconnected, a contradiction.

□

The local intersection path. The ***local intersection path*** beginning at the gluing vertex $\hat{v}$ is defined by walking along the intersection between one face at a time from R_1 and one face at a time from R_2 at a time starting at $\hat{v}$. When the walk encounters an edge of a face, it traverse across it into the next face in that

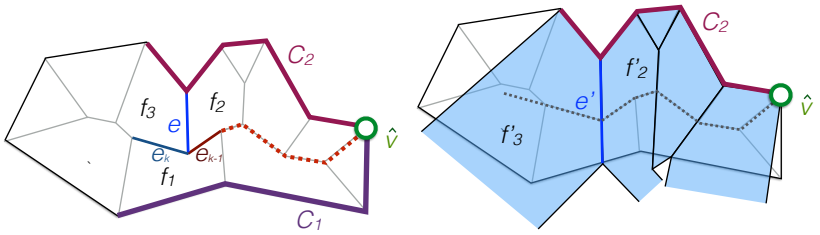


Figure 4.8: An illustration of the inductive proof of Lemma 4.4.2. The left shows the critical path (dotted) leading up to edges e_{k-1} and e_k. The chains C_1 and C_2 are drawn as thick (purple and maroon) lines. The right shows a partial roof for C_2. Since f_2' and f_3' satisfy face containment and edge existence and containment there must be an edge e' between them that contains e and so the local intersection path between f_1' and f_2' will hit e' at pass in to f_3'.

partial roof. For now we assume that the walk always hits an edge and not a vertex, though we remove this assumption at the end of this section. Note that if the two slabs share a common motorcycle edge, then we treat this as an intersection of the two. It is helpful, however, to distinguish between edges formed along a common motorcycle edge between two slabs, and edges formed along the normal intersection between two slabs. We will call the first ***valley candidates*** and the second ***ridge candidates***. The most important feature of the local intersection path is given by the following lemma which is proved by induction along the path. We illustrate the induction in Fig. 4.8, but leave the full proof to the appendix. The main idea is that by face containment an edge of the final straight skeleton roof on the critical path is necessarily contained in the intersection of the faces of the two partial roofs, and as we walk along this intersection. If there is a next edge along the critical path then by the edge existence and containment properties, the local intersection path must hit this edge. The rest follows by induction along the path. See Fig. 4.8 for a visual overview.

Lemma 4.4.2. *The critical path is a connected subset of the local intersection path from $\hat{v}$.*

Proof. Let e be an edge of the critical path. Let s_1 be the slab from S_{C_1} and s_2 be the slab from S_{C_2} supporting e. Let f_1' and f_2' denote the faces of R_1 and R_2 (resp.) supported by s_1 and s_2 and let f_1 and f_2 be the corresponding faces in the final roof R_P. $\Diamond$: We first argue that $e \subseteq f_1' \cap f_2'$. Assume not. Since $f_1 \subseteq f_1'$ and $f_2 \subseteq f_2'$ (by face containment on R_1 and R_2), then e is not a subset of $f_1 \cap f_2$, a contradiction.

We now prove the lemma by induction. Assume the lemma holds for the first $k-1$ edges of the critical path. Let e_{k-1} and e_k denote the $(k-1)^{th}$ and k^{th} edges of the critical path and e'_{k-1} denote the $(k-1)^{\texttt{th}}$ edge of the local intersection path. We first argue that $e_{k-1} = e'_{k-1}$. We prove that it holds for the $k^{\texttt{th}}$ edge. Let e_{k-1} and e_k be the $(k-1)^{\texttt{th}}$ and $k^{\texttt{th}}$ edges of the critical path. By the inductive hypothesis the $(k-1)^{\texttt{th}}$ edge of the local intersection path e'_{k-1} contains e_{k-1} and both start at the same vertex u. Let v be the vertex between e_{k-1} and e_k. Generically, there are three faces in the final straight skeleton roof R_P incident to v, one will be incident to both e_{k-1} and e_k. Call this f_1. The other two, f_2 and f_3, are incident to e_{k-1} and e_k (resp.). *Wlog* assume f_1 has its base edge in C_1 and f_2 and f_3 have base edges in C_2. Let f_1', f_2', and f_3' be the corresponding faces of R_1 and R_2. See Fig. 4.8. We first note that the edge e between f_2 and f_3 incident to v is a critical edge for R_2. Thus f_2' and f_3' have an edge between them e' and $e \subseteq e'$ (edge existence and containment on R_2). Thus, by definition of the local intersection path, e_{k-1} "hits" e' at v. The next edge of the local intersection path is then between f_3' and f_1'. Call this edge e'_k. It remains to show that $e_k \subseteq e'_k$. Assume not. Both e'_k and e_k start at v and lie along the intersection of the slabs supporting f_1' and f_3', so this implies that $e_k \not\subseteq f_1' \cap f_3'$, which contradicts $\Diamond$. Below we define the local intersection path for non-generic cases (see the paragraph titled "Handling Degeneracies"). With our definition, it is straightforward to extend the argument above to generic cases by noting that the if you look at the fan of faces around v in R_P, all of its incident edges that are not part of the critical path are critical in R_1 or R_2. $\square$

The splicing path The significance of Lemma 4.4.2 is that in order to ensure that we detect all of the edges required to satisfy edge containment, it suffices to use the local intersection path. The problem with this approach, however, is that if we use the entire local intersection path in the merge operation detailed in the next section, we cannot guarantee the maintenance of certain properties of the output (like the fact that it is topologically a disk). Instead we use a subchain of the local intersection path that starts at $\hat{v}$, which we call the *splicing path*. It is defined as follows.

Definition 4.4.3. *The* ***splicing path*** *is the longest subchain of the local intersection path starting at $\hat{v}$ satisfying the following properties.*

1. *The path* visits *each face of R_1 (resp. R_2) at most once. This includes disallowing the splicing path to touch the boundary of the face without leaving it.*

2. *The part of the path lying on the same slab s can be decomposed into at most three distinct subchains: a chain of* ***valley candidates*** *along one motorcycle arm chain of s, a chain of* ***ridge candidates*** *that lie on the intersection of s with other slabs, and a second chain of valley candidates along the other motorcycle arm chain of s.*

3. *For each edge e, if it is a ridge candidate then the slab from R_1 (resp. R_2) containing e slopes* downwards *to its left (resp. right)*[3]*. If it is a valley candidate, then the slabs slope* upwards *to the left (resp. right).*

These properties constitute a set of necessary conditions for an edge of the local intersection path to be an edge of the critical path (the reader should compare the properties in the definition above to those in Lem. 4.4.1), and so we have:

Lemma 4.4.4. *The splicing path contains the critical path.*

[3] *Right* and *left* for the edge e are defined by directing e away from the gluing vertex $\hat{v}$ and projecting e downwards into the xy-plane, giving a well-defined right and left side.

Handling non-genericity In our definition of the local intersection we assumed the input was generic, which in our case meant that the local intersection path never intersects a vertex of either R_1 or R_2. This ensures that the local intersection path is well-defined. If the path does hit a vertex v of, say, R_1 then it may there may be more than one possible "next" edge between face of R_1 incident at v and the current face in R_2. The problem is compounded if the local intersection path hits an vertex both in R_1 and in R_2 simultaneously; however, now it should be clear that the point of walking along the local intersection path is to ensure that we pick up all of the edges of the critical path. This allows us to unambiguously define the local intersection path in non-generic situations–the path we define will contain the critical path. Let v be a vertex on the critical path in R_P incident to more than three faces and e_1 and e_2 be the incident critical edges (where e_1 comes before e_2). Let F_L be the fan of faces on the left side of e_1, e_2 and F_R be the fan on the right side. Necessarily, faces of F_L all have their base edges in C_1 and the faces of F_R all have their base edges in C_2 (property 2 of the critical path). Furthermore, the edges between faces in F_L are critical for R_1 and those in F_R are critical for R_2. Now, suppose at some point along the splicing path we walk along an edge e'_1 corresponding to e_1 and hit a degenerate vertex v' (we will assume we simultaneously hit a vertex in both R_1 and R_2, since this is the harder case). Following the same argument as in Lemma 4.4.2 we have that $v = v'$. Now let F'_L be the faces incident to v' in R_1 ordered clockwise around v' starting from e'_1 and let F'_R be the faces incident to v' in R_2 ordered counter-clockwise around v'. Let e'_l be the first intersection between F'_L and F'_R in clockwise order in F'_L and e'_r be the first intersection between F'_L and F'_R in counterclockwise order around F'_R. Then we have:

Lemma 4.4.5. *$e'_l = e'_r$ and this intersection contains e_2.*

Proof. Assume not. By face containment we have that (1) the first faces in the clockwise ordering of F'_R correspond to the faces of F_R (similarly with F'_L and F_L in

the counter-clockwise ordering and (2) there exists an intersection e' between F'_L and F'_R that corresponds to e_2. Our assumption implies that there must be an intersection e'' before that in (*wlog*) the clockwise ordering of F'_L. This necessarily lies on one of the faces f' of F'_L that corresponds to a face f of F_L. The two edges incident to v in f are critical edges, and thus must exist in f'. This means e'' lies between two critical edges of a face, and so in the local neighborhood of f at v there is some intersection between f and a slab between its two critical edges. But the existence of this intersection contradicts that f is part of the lower envelope. □

In other words, the first intersection between F'_L and F'_R in their respective orderings is between the slabs supporting e_2. This is unambiguous and contains the next critical edge, so we define this to be the next edge of the local intersection.

To compute this intersection, we restrict each fan F'_L and F'_R to the faces making a total angle sum not greater than 2π (since necessarily, if a critical edge e_2 exists past v, the angle between e_1 and e_2 in both R_1 and R_2 must be less than 2π). Think of intersecting a vertical cylinder with sufficiently small radius centered at v with each fan. The intersection of each fan with the cylinder is a chain of curved edges along the surface of the cylinder. If the chains are not monotone, simply discard anything past the point of monotonicity (these cannot be part of the lower envelope and therefore cannot contain e_2). Now we have two monotone chains described on a cylinder, and we are simply looking for the first intersection of one with the other, which can easily be found in time linear in the degree of v.

4.4.2 The merge operation

We now describe how to merge R_1 and R_2. First walk along the local intersection of R_1 and R_2 starting at $\hat{v}$ while maintaining the properties of Def. 4.4.3 in order to obtain a splicing path p (as defined in Sec. 4.4.1). Next, cut each face traversed by the splicing path and discard the part of the face that lies above the splicing path

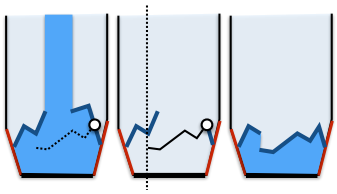


Figure 4.9: The cut and discard step. The splicing path enters the face at the white circle and we remove the rest of the monotone chain from that point. In this example the splicing path stops on the interior of the face, and so check that the two upper monotone chains are co-monotone. If not, we truncated the other to maintain co-monotonicity. The final face is shown on the right.

(some care must be taken with the end of the splicing path, which generically stops on the interior of a face). The resulting structures still satisfy face containment and the edge existence and containment properties (Lemma 4.4.6). Both are topological disks (Lemma 4.4.7) and the splicing path is now part of the boundary chain of each. Next, perform a cleanup step to remove certain *orphaned* edges. Finally, glue the two surfaces together along the splicing path to obtain a partial roof R for C (Cor. 4.4.10).

The cut-and-discard step Let f be a face of R_1 and $\texttt{sp}_f$ be the chain or ridge candidates of the splicing path across f. By our convention, the downward slope from each ridge candidate along $\texttt{sp}_f$ will be to the left (since we are in R_1), meaning that the splicing path moves from right to left (monotonically) across each face. Let p be the point at which $\texttt{sp}_f$ enters f. Perform the following

1. If f is unbounded, remove the part of the motorcycle arm chain and upper monotone chain of f that is connected to p that does not contain the base edge of f. See Fig. 4.9.

2. Add the edges of $\texttt{sp}_f$ as part of the upper monotone chain starting at p.

3. If $\texttt{sp}_f$ hits the motorcycle arm chain or upper monotone chain on the other side of f at a point q, discard the part of the motorcycle arm chain and upper monotone chain that is monotonically above $\texttt{sp}_f$. Otherwise, if $\texttt{sp}_f$ ends on the interior of f, check if it passes monotonically below the upper monotone

chain on the other side of f. If it does, remove the part of that chain that is monotonically above $\mathtt{sp}_f$. See Fig. 4.9.

Let R_1' and R_2' be the disks resulting from this operation. We prove:

Lemma 4.4.6. *R_1' and R_2' satisfy the face containment, edge existence and edge containment properties for their base chains.*

Proof. We first argue that if f is unbounded, the chain of edges discarded by step 1 are not critical. Call this the ***discarded chain***. Since $\mathtt{sp}_f$ has an edge on the interior of f incident to p and each edge of $\mathtt{sp}_f$ lies on the intersection of slabs, then the part of the discarded chain immediately adjacent to p cannot be on the lower envelope of slabs, meaning that the part of the discarded chain immediately incident to p is not critical. If any edge of the discarded chain is critical, then we have a contradiction of Lemma 4.3.5. The remaining steps essentially discard the parts of the face that lie monotonically above the $\mathtt{sp}_f$. Thus, anything discarded cannot be part of the lower envelope, and therefore is not critical or needed to satisfy face or edge containment. □

We note that topologically, if $\mathtt{sp}_f$ crosses f, then we are essentially cutting f into two disks and discarded the part on the right side of $\mathtt{sp}_f$ (in R_1, or left side in R_2). The splicing path necessarily crosses all but the last face it enters. In the last face, topologically speaking, we are removing a disk from f that is incident along the entire right side of $\mathtt{sp}_f$ and the boundary of f at p. We now show that when we discard all of these disks, the resulting surface R_1' (resp. R_2') is itself a disk.

Lemma 4.4.7. *R_1' (resp. R_2') is a topological disk.*

Proof. By definition, the splicing path is composed of two types of edges, ridge candidates, which lie along the proper intersection of faces, and valley candidates which necessarily lie along motorcycle edges of slabs, meaning that these edges are along the

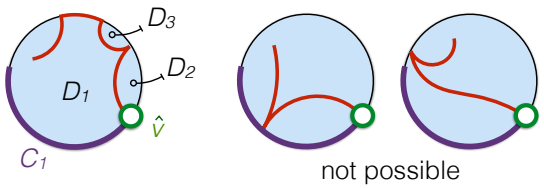


Figure 4.10: *Left:* an illustration of the topology of the splicing path–it divides the surface into several disks, one of which must contain the entire base chain. *Right:* two examples of splicing path topologies that are not possible–one because it intersects the base chain leading to a contradiction, the other because after touching the boundary, it turns back onto the side not containing the base edge, which implies that it must visit the same face twice.

boundary of R_1. Non-generically, the splicing path may also pass through a boundary vertex of R_1. Thus the splicing path can be decomposed into connected chains of edges starting at the gluing vertex $\hat{v}$ that pass through the interior of R_1 (resp. R_2) to another boundary point or boundary edge. Cutting along this path necessarily results in a series of disks $D_1, \ldots, D_k$. See Fig. 4.10. *Wlog* let D_1 be the disk incident to $\hat{v}$ that contains the edge of C_1 incident to $\hat{v}$. We are going to show that the discard step discards all of the faces in $D_2, \ldots, D_k$, and that after discarding all faces lying to the right side of the path in D_1, it remains a disk.

We first argue that D_1 contains the entire base chain C_1. Assume not. Then the splicing path must cut through the base chain (since we only discard parts of a face monotonically above the splicing path with respect to its base chain). But each slab (and therefore each face) is incident to the xy-plane only along its base edge, so this implies an intersection between base edges, a contradiction. We next argue that $D_2, \ldots, D_k$ are all incident along the right side of the splicing path. Assume not. The disk D_1 is immediately incident to the left side of the splicing path at $\hat{v}$ (since it contains the base edge in C_1 incident to $\hat{v}$, which by our convention is on the left of $\hat{v}$) so for any other disk to be incident to the left side of the splicing path, the splicing path must cut all the way across R_1 to create D_1. See Fig. 4.10. By property 1, the splicing path cannot hit the boundary of a face without exiting it, so this means the

next edge of the splicing path is in some face not yet traversed by the splicing path. But this face must contain a base edge, which is not part of D_1, a contradiction.

Therefore, $D_2, \ldots, D_k$ are all to the right of the splicing path. We now argue that *all faces* in D_2 through D_k are incident to the splicing path and are hence discarded by the discard step. Assume not. Then there is some face f in, say, D_2 such that it is not incident to the splicing path. But this face is then left intact by the cut step, meaning it is not traversed by the splicing path. Therefore it must then contain a base edge, a contradiction. Therefore the cut-and-discard step removes $D_2, \ldots, D_k$.

Now we have that $D_2, \ldots, D_k$ are discarded, but the splicing path may not cut all the way through R_1, which in particular means it ends somewhere on the interior of D_1. In general, if D_1 was any disk and we took a path from its boundary that stopped on its interior and discarded the faces to the same side of the path, we might disconnected D_1 into multiple disks. We show that this does not happen in our case. Suppose it does. Then there is some face f that is disconnected from the rest of D_1 by removing the faces along the right side of the splicing path in R_1. Because f is not discarded it is not traversed by the splicing path and therefore must contain a base chain. Thus the base chain is disconnected. But, as we saw before, we only discard the parts of faces above the splicing path, so this implies the splicing path intersects the base chain, a contradiction. □

Cleaning up When we discarded edges in the previous step, we did so in a single face, but each edge "discarded" is really an internal edge between two faces. Discarding only removes the edge from one of the faces, thus making it a boundary edge in the other. For example, if e is an monotone chain edge between faces f_1 and f_2 and the splicing path passes beneath e in f_1, then e is "discarded" from f_1 but remains in f_2, thus becoming a boundary edge in R_1'. See Fig. 4.7. The face construction property, however, does not allow for such edges. Call these ***orphaned***

edges. The following lemma implies that if we simply remove these edges we maintain the face containment property. We leave the proof to the appendix.

Lemma 4.4.8. *The orphaned edges of a face f of R'_1 (resp. R'_2) are connected on the boundary of f, and if f is unbounded are at the end of a monotone chain (i.e. end at its free point).*

Proof. Assume not. First assume f is bounded. Then there is some non-orphaned edge e of the monotone chain between two orphaned edges. Recall that the orphaned edges are boundary edges. There must be a second face f' incident to e (otherwise it would be orphaned). But then any path along the boundary from the base chain of f' to the base chain of f must pass through an orphaned edge of f, contradicting that C_1 is connected along the boundary of R'_1 (resp. R'_2). Now assume f is unbounded. Recall that the unbounded portion of the face is formed by two edges along f incident to a vertex at infinity that are part of the boundary of R_1. We first argue that all orphaned edges are incident to the same monotone chain. Assume not, then there is an orphaned edge in each monotone chain of f. Let e_1 be an orphaned edge on one and e_2 be an orphaned edge on the other and let f_1 and f_2 be the faces that were cut by the splicing path to orphan e_1 and e_2. By the definition of cut-and-discard, we have that the splicing path cannot pass beneath the unbounded part of f. But if we remove f from R_1, it necessarily disconnects f_1 from f_2, which means that the splicing path is disconnected, a contradiction. □

We now cleanup by removing all orphaned edges from a face. We have already seen that none of these edges are needed by the edge existence property (otherwise they would not have been orphaned). Removing them from a bounded face simply cuts its monotone chain into two, and removing them from an unbounded face simply truncates a single monotone chain (by Lemma 4.4.8). Therefore this step does not

discard anything we need, and only increases the size of each face. Let R_1'' and R_2'' be the resulting surfaces. We have:

Corollary 4.4.9. *R_1'' and R_2'' are topological disks and satisfy the face containment, edge containment, and edge existence properties for their base chains.*

Gluing R_1'' to R_2'' Finally, we glue R_1'' to R_2'' along the splicing edge, which is now a boundary path in each to obtain R. R_1'' and R_2'' are topological disks that satisfy all the partial roof properties on their respective chains C_1 and C_2 except for face construction, since the splicing path edges are boundary edges (and thus not internal edges). The splicing path necessarily contains the critical path. Thus, if we glue R_1'' to R_2'' by identifying corresponding edges along the splicing path to obtain R, then the splicing path edges become internal edges in R. Since the splicing path necessarily contains the remaining critical edges between faces in R_1 and faces in R_2, the cleanup step got rid of any orphaned edges that violated face containment, and gluing along the boundary maintains the geometry of each face and edge in R_1'' and R_2'', we have:

Corollary 4.4.10. *R is a partial roof for $C_1 \oplus C_2$.*

Putting everything together, we arrive at the main result of this section:

Lemma 4.4.11. *Given valid partial roofs R_1 and R_2 for subchains C_1 and C_2 as input, there is an algorithm for computing a partial roof R for $C_1 \oplus C_2$.*

4.4.3 Complexity

We represent a partial roof by storing, for each of its slabs s, the lower convex chain and left and right upper monotone chains as doubly-linked lists of half-edges (in a similar vein to the DCEL data structure [35]). Each half-edge on the upper monotone chain(s) stores a pointer to its corresponding edge in the other face that it is incident to. Most of the operations are straightforward in this representation. The

main difficulty is computing the splicing path, which involves repeatedly computing the intersection of two faces. We first decompose each face into trapezoids by adding edges from each vertex along the direction of monotonicity. Since each face is monotone, decomposing a face takes linear time. Instead of intersecting faces, we intersect trapezoids, which is a constant time operation. A face is given by one or two sets of trapezoids and an unbounded trapezoid if the face is unbounded. We walk across the intersection of two faces by walking across the intersection of the corresponding trapezoids. Because the splicing path is monotone across each face, we visit each trapezoid at most once. Thus, the number of edges in our trapezoid splicing path is linear in the number of trapezoids. To show that the operation takes linear time in the complexity of the partial roof, we bound the number of trapezoids created. Each vertex of a face is incident to at most two trapezoids in that face. Thus the total number of trapezoids created is twice the sum of the degrees over all vertices. In generic cases each time we compute an intersection between trapezoids it takes $O(1)$ time to traverse to the next trapezoid, however, in non-generic cases where the intersection path hits a vertex we use the method from Sec. 4.4.1. Since this method is linear in the degree of hte vertex, then the worst case running time over all edges is the sum of the degrees. By a trivial application of the handshaking lemma and by Lemma 4.3.4, we have:

Lemma 4.4.12. *Let R_1 and R_2 be partial roofs for subchains C_1 and C_2 such that S_{C_1} and S_{C_2} have k_1 and k_2 base and motorcycle edges. Then the merge operation takes $O(k_1 + k_2)$ time.*

Our divide and conquer algorithm is this: subdivide the polygon into equal length chains C_1 and C_2, recursively compute a partial roof for each and merge the result. Thus we have:

Theorem 4.4.1. *The straight skeleton of a polygon P with n edges can be computed from its motorcycle graph in $O(n \log n)$ time.*

Proof. Correctness of the algorithm follows from Lemma 4.4.11. In non-degenerate cases the number of motorcycle edges in a slab is $O(1)$, Lemma 4.4.12, so the length of the subchain $C_1 \oplus C_2$ is $O(k_1 + k_2)$ and the analysis is the same as merge sort. The running time of each merge step is linear in the number of trapezoids generated, and the endpoint of each edge of a face accounts for two trapezoids in the decomposition of the face. Thus each motorcycle edge of a slab "contributes" $O(1)$ trapezoids to each merge step in which the slab appears. Each slab appears in only $O(\log n)$ merge operations and in degenerate cases there are a total of $O(n)$ motorcycle edges. Thus, by amortized analysis in the degenerate case the operation takes $O(n \log n)$ time. □

4.5 Generalizing to PSLGs

Our algorithm depends on the fact that the new edges we need to pick up at each merge step correspond to a path in the final straight skeleton which in turn are contained in the local intersection path of the two input partial roofs. This is due, mainly, to the fact that the straight skeleton of a polygon contains no interior faces and the base chains of the two input roofs are connected at the gluing vertex.

In the general case of a PSLG or a polygon with holes, the "boundary" is now the set of connected components. In particular, this means that the base chains are no longer connected. In order to extend the algorithm we compute a subdivision of the plane in the case of a PSLG G or the interior of the polygon P (in the case of a polygon with holes) into polygonal cells such that each polygonal cell contains no entire face of the straight skeleton on its interior. The boundary of each cell is treated at a set of base edges, which allows us to employ our polygon algorithm (with slight modifications).

We employ the subdivision algorithm of Cheng et al. [24] to compute the subdivision. We note that they only define the subdivision for polygons with holes, but their method (and proofs) extends naturally to the outer face of a PSLG. The algorithm

is a recursive one. It maintains a subdivision of the interior of the polygon into cells and each recursive invocation takes one of the cells and further subdivides it. The subdivision is performed on the set of vertices in the induced motorcycle graph which has $O(r)$ complexity and leads to a recursion depth of $O(\log r)$. This is necessary for [24], because they require a stricter set of properties of the final subdivision than we do. Namely, they require that the part of the straight skeleton above each cell be convex. In our case, we relax this requirement. We require only that no face of the straight skeleton is interior to a cell. Instead of subdividing on all $O(r)$ vertices in the induced motorcycle graph, we instead subdivide on one reflex vertex per connected component (or hole), leading to a recursion depth of $O(\log m)$, where m denotes the number of connected components in the PSLG (or holes in the polygon). Since $m = O(r)$ but it is not necessarily the case that $r = O(m)$, this improves on the algorithm.

4.5.1 Subdivision Algorithm

To create the subdivision we use the cellular subdivision algorithm of Cheng et al. [24]. Before we review their algorithm, we need two subroutines, one computing the intersection of the straight skeleton roof with a vertical-plane and one for computing descent paths. We then give a brief overview of the algorithm and define its main data structure.

Vertical-plane intersection subroutine. We call a plane Π that is parallel to the yz-plane a ***vertical plane***. The vertical-plane intersection procedure computes the intersection of Π with the straight skeleton roof R_P of a polygon P (or, similarly, the straight skeleton terrain R_G of a PSLG G) *without first computing the entire roof.* First, intersect each slab in slabs(P) with the plane Π. The intersection of each slab is a line segment in Π. Denote the resulting set of line segments by S. Since R_P is the lower envelope of slabs, it follows that the lower envelope of the set of line segments S

is equivalent to the intersection of Π with R_P. This can be computed using standard sweep techniques in $O(n \log n)$ time where n denotes the number of segments[4] [36].

Descent path subroutine. Recall that given a point p on the roof, a ***descent path*** is the path of steepest descent downwards from p. If p is on the interior of a face, then the descent path starts with a segment downwards along the slope of the face. This segment either hits the base edge of the face and thus the descent path is just the one segment, or hits a motorcycle edge of the face. Since the motorcycle edges are valleys in the terrain, the descent path then follows the motorcycle edge back down to the base. In particular, this means that to compute the descent path for a point p on a face f, we do not need to have an explicit representation of f. Rather, we need only the representation of the slab s supporting f. If a slab has m motorcycle edges incident to it, then computing a descent path for a point p on the face supported by f requires $O(m)$ time. In particular, in generic cases, this means that computing a descent path takes $O(1)$ time. Note that if p lies on an edge of the straight skeleton, then it has two decent paths, one in each face incident the edge. If p lies on a vertex of the straight skeleton, then there is a descent path for each face incident the vertex, which is 3 in generic cases.

Overview of the subdivision procedure. We now review the subdivision procedure. We first describe it for a polygon with holes, and then describe how to modify it for a PSLG. Let P be a polygon with holes. The subdivision algorithm computes a subdivision of P into cells $C_1, \ldots, C_l$.

---

[4]In the case of a PSLG, the intersection of a slab with Π may be either a line segment or a ray. The problem of computing the lower envelope of a set of rays and segments is easily reducible to that of just segments. First, in $O(n \log n)$ time compute a bounding box of the intersections of the supporting lines of the rays and segments (this is possible since an extreme point of the arrangement of n lines always lies on the intersection of lines with consecutive slope, cf. [58]). Then clip each ray to the boundary of the bounding box to form a set of segments. Next, compute the lower envelope of the resulting segments. Outside the bounding box no rays can cross, and computing the remaining parts of the lower envelope is trivial.

The cell data structure. Each cell C_i is a polygon drawn on P. Its edges may lie along the boundary of P or may be on its interior. The algorithm maintains a lifting of C_i onto the straight skeleton roof R_P, denoted $\hat{C}_i$. Each edge of a cell lies on only one face of R_P, meaning in particular that each edge e has a lifting $\hat{e}$ onto a line segment drawn on a single face of R_P. For each edge e we store a pointer to the slab s which supports $\hat{e}$ in R_P. A cell also contains a *slab list* slabs(C_i) which is equivalent to the slabs defined for each of its edges along with the slabs for any edge of a hole contained on the interior of slabs(C_i). The subdivision procedure maintains the invariant that the part of the roof R_{C_i} above C_i is the lower envelope of slabs(C_i) (cf. [24]).

To handle PSLGs, we need to store unbounded cells. An unbounded cell is bounded by two rays in the plane connected by a polygonal chain. We handle this by adding a single vertex at infinity between the two rays (making each unbounded cell a cycle combinatorially) and storing a direction at the two rays.

Vertex conflict list. Each cell C_i additionally maintains a certain set of vertices from the polygon P that lie on its interior, which we call its ***vertex conflict list*** and denote by V_i. We will see later how to initialize and maintain this list.

Vertical partitioning of a cell. The main subdivision procedure takes as input a cell C_i and a vertex v of the vertex conflict list V_i and *vertically partitions* C_i via the following procedure. Let $v \in V_i$. Let Π be the vertical plane through v. First, compute the vertical plane intersection (using the subroutine defined above) between Π and R_{C_i}. Let p denote the polygonal path along the intersection of Π with R_{C_i}. Each edge of p is labeled with the slab of slabs(C_i) supporting it. Each internal vertex along p is either on some part of the boundary of P that is on the interior of C_i or lies on a straight skeleton edge of R_{C_i}. In the latter case, we trace descent paths downwards from the vertex. The projection of the descent paths and p down onto the xy-plane further subdivides C_i into at least two cells (on either side of the vertical

plane). The conflict list V_i is then split between these cells–each vertex in V_i is placed in the conflict list of the sub-cell that it is contained in. If a conflict vertex is now on the boundary of a cell, we discard it.

One important property of this procedure is that because the conflict vertex v lies on the boundary of P but not on the cell boundary, it means that each time we apply the algorithm, a hole of the polygon becomes part of a cell boundary. With the appropriate initial conflict list, this property allows us to guarantee that no hole of P lies on the interior of any cell in the subdivision procedure.

Subdivision procedure. Let V denote a set of vertices, one from each hole in P (or connected component in the case of a PSLG), such that each vertex is reflex. We apply the vertical partitioning procedure above recursively. Initially we have one cell $C_0 = P$ with V as its vertex conflict list. Our choice of V is a strict subset of the set Cheng et al. use in their subdivision algorithm, which allows the relevant proofs to carry through without modification. At each step we take a cell C_i such that its slab set has at least three slabs and it has at least one vertex in its conflict list. We then select the vertex from the vertex conflict list having the median x-coordinate, and apply the vertical partitioning procedure on it. This results in a further subdivision of the cell C_i which we recursively subdivide further, so long as the subdivided cell has at least three slabs and one conflict vertex.

Properties of cells. The subdivision algorithm guarantees several important properties for the final subdivision $C_1, \ldots, C_l$:

1. Either slabs(C_i) is ***empty***, meaning that the cell C_i lies entirely in one face of the straight skeleton; or slabs(C_i) contains exactly two slabs (Cheng et al. call this a ***wedge***); or slabs(C_i) is ***non-trivial*** but contains no connected component of P (i.e. no hole) on its interior. [24, Lemma 14].

2. The lower envelope of slabs(C_i) is R_{C_i}.

3. No face of the final straight skeleton roof lies entirely on the interior of a cell ([24, Lemma 14]).

4. Each slab appears in the slab set of $O(\log m)$ cells and $\sum_i |\operatorname{slabs}(C_i)| = O(n \log m)$ ([24, Lemmas 8 & 13]).

5. Let C_i be a non-trivial cell and f be a face of R_{C_i}. The edges of C_i incident to f form a (connected) lower convex chain in the slab supporting f. The endpoints of this chain are either on ridge edges of R_{C_i} in which case the face f lies monotonically above the chain. Or the endpoints lie on a motorcycle edge of R_{C_i} in which case f lies monotonically above the chain and its incident motorcycle edges.

Furthermore, since we perform exactly the same recursion, but on an $O(m)$ sized subset of the $O(r)$ vertices used in [24] (where m denote the number of holes in P), the same analysis as in [24, Lemma 14] shows that the subdivision runs in $O(n \log n \log m)$.

Property 5 allows us to define a ***subdivided slab*** for each chain of edges in C_i with the same supporting slab s–simply restrict s to the region above the chain. This is a generalization of the slabs we use to compute the polygon straight skeleton. Now a slab may be incident to a connected chain of edges in C_i rather than a single base edge. Because of 5, the lower envelope of the subdivided slabs for C_i is R_{C_i}. We now run our algorithm for computing the straight skeleton of C_i with the following modification. Instead of splitting C_i into sub-chains, we only split between edges that are supported by different slabs of C_i. When we split, we never split a subdivision boundary chain. The subdivision boundary chains now play the role of the base edge in the original algorithm. The direction of monotonicity for the splicing path is checked against the original base edge's monotone direction and not the subdivision boundary chain. Since our algorithm fills in a cell C_i in $O(|\operatorname{slabs}(C_i)| \log |\operatorname{slabs}(C_i)|)$ time (amortized in degenerate cases) and $\sum_i |\operatorname{slabs}(C_i)| = O(n \log m)$ and the cellular

subdivision can be computed in $O(n(\log m)\log n)$ time, then the entire algorithm takes $O(n(\log m)\log n)$ time, and we have:

Theorem 4.5.1. *The straight skeleton of a PSLG with n edges and m connected components can be computed from its motorcycle graph in $O(n(\log m)\log n)$ time.*

.

4.6 Conclusion

We have presented an algorithm for computing the straight skeleton given its motorcycle graph as input that speeds up the computation of the straight skeleton in all cases. One of the main novelties is relaxing the requirement that intermediate structures computed by the algorithm be either terrains or lower envelopes. Instead, our lower envelopes are neither, even though they are topologically disks. It remains open, however, how fast the straight skeleton can be computed, and more work is needed to either improve the algorithm or the known lower bounds.

4.7 Future work

The algorithm we present speeds up the computation of the straight skeleton in all cases. One particularly useful result of this is that computing the straight skeleton of PSLGs now follows the same scheme as that of polygons and polygons with holes; namely, we first compute the motorcycle graph and then compute the straight skeleton. Thus, given our algorithm, the bottleneck in all cases now becomes the motorcycle graph (recall that before, in the PLSG case the bottleneck for algorithms that computed the straight skeleton from the motorcycle graph was the second part of the computation). Thus, it remains an interesting open problem what is the fastest possible algorithm for computing the motorcycle graph.

CHAPTER 5

THE UNIVERSAL MOLECULE

We now investigate the properties of the universal molecule, which is (in some sense) a generalization of the straight skeleton for convex polygons. Prior to the present work, the universal molecule was a heuristic, in that a clear proof of correctness had not yet appeared. This was in part due to the lack of a characterization (independent of the algorithm itself) of precisely what objects the algorithm computes. A main contribution of this chapter is that we develop a family of piecewise-linear surfaces we call ***Lang surfaces*** (surfaces formed by gluing polygonal faces together along whole edges) constructed on top of trees using two simple operations. We show when we restrict this family of surfaces to have zero-curvature and convex boundary, the surfaces are exactly the set of surfaces that are (intrinsically) computed by the universal molecule algorithm. Once we have established this characterization, we prove the correctness of Lang's universal molecule algorithm as it was originally formulated. This work has appeared as [15]. Having this characterization opened up new insights to the universal molecule algorithm. For instance, it enables us to generalize the algorithm in Ch. 7. There, we use Lang surfaces to generalize the universal molecule algorithm to non-convex polygons. This characterization also allows us to analyze the rigidity properties of the universal molecules in Ch. 8.

5.1 Introduction

We now briefly review Lang's origami design problem, TreeMaker, and the universal molecule algorithm which was presented in Ch. 2.

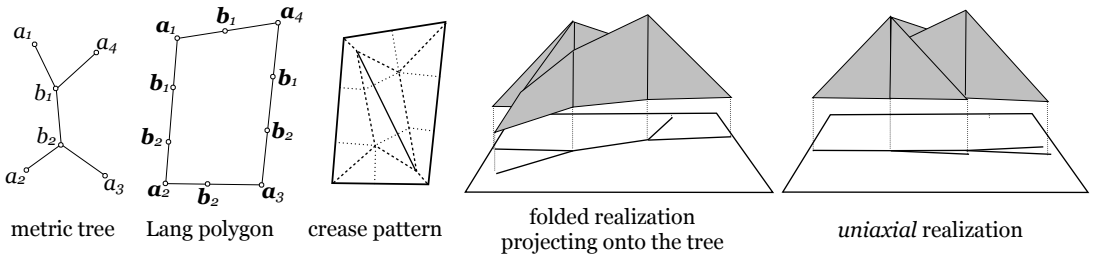


Figure 5.1: The structures involved in the universal molecule algorithm.

Lang's Origami Design Problem. The universal molecule algorithm is a component of the TreeMaker method for origami design proposed by Robert Lang in 1996 [44] and implemented in his freely available software [45]. Starting with a piece of paper and a tree-like sketch of what the final folded shape should look like (as in Fig. 5.1(left)), TreeMaker tries to produce a crease pattern with a guaranteed folded 3D shape resembling the given tree. This shape can be flattened into a single plane so that its boundary edges all lie along a single line or *axis*; Lang refers to this as a *uniaxial base*. This informal definition for *uniaxial* is quite broad and encompasses many different 3D shapes, not all of which are produced by the TreeMaker algorithm.

TreeMaker works in two phases, both of which raise interesting, yet insufficiently investigated theoretical questions. The *first phase* is an optimization procedure which subdivides the paper into polygons while minimizing a rescaling factor. If some face in the subdivision is non-convex, TreeMaker stops with an error message. Understanding what conditions result in non-convex faces and modifying the optimization step to guarantee convexity remain open questions which we do not address. If all the resulting faces are convex, then the *second phase* fills them in with creases using the *universal molecule (UM) algorithm*. The interior of each polygon folds along the creases in such a way that it projects to some portion of the tree. Lang describes the crease pattern and folded shape resulting from applying the UM algorithm on a convex polygon, but, due to the lack of precision in some of his definitions, it is difficult to assess the overall correctness of the method. In particular, it is not clear

a priori what the *precise relationship is between the input to the algorithm (the tree and polygon) and the final 3D folded shape.*

In this chapter we focus on the universal molecule (UM) algorithm as it is applied in Lang's original context–namely, to convex polygons. Our first contribution is to unambiguously state the problem, and to clarify the specific relationship that must exist between the tree and the convex piece of paper on which the crease pattern will be designed. Precise definitions are reviewed in Section 5.2, but here is a preview: the *input* is a pair consisting in a metric tree T and a *doubling polygon* P_T for T (See Ch. 2). If the Euclidean distance between any pair of polygon vertices is at least as large as the tree distance between the corresponding tree leaves, the doubling polygon is called a ***Lang polygon***. The *output* is referred to as a zero-curvature ***Lang surface*** constructed on T with convex boundary. It has a 3D realization which projects onto the input metric tree T. Fig. 5.1 illustrates the concepts. The main contribution of this chapter is to prove:

Main Theorem. *Let T be a metric tree and P_T be a convex doubling-polygon associated to it. Then a Lang surface S constructed on T and isometric to P_T exists (and is unique) if and only if P_T is a Lang polygon for T.*

The necessity of Lang's property, already implicit in [44], is not hard to prove. The proof of sufficiency, which proceeds via Lang's Universal Molecule algorithm, occupies most of the chapter.

Divide-and-conquer Parallel Sweep. Lang's algorithm resembles the parallel sweep of the straight skeleton (See Ch. 2), in that it works by moving the sides of the polygon inwards in a parallel fashion at unit speed. The *universal molecule crease pattern* is obtained by tracing the vertices of the sweeping polygon. Novel to the universal molecule algorithm is a simultaneous shrinking process in the metric tree. Specific invariants maintain relationships between the sweeping polygon and

the shrinking tree. Events occur when the invariants are violated, and are of two types: *contraction events,* in which edges of the polygon and arcs of the tree shrink to zero length, and *splitting events,* in which the polygon (and tree) is split into two and the sweep continues, recursively, on each side of the split. A contraction event leads to a polygon with a smaller number of vertices, and a splitting event leads to two smaller polygons, on which the algorithm proceeds recursively. Lang's algorithm is thus a mixture of parallel sweep and divide-and-conquer paradigms, in that there is a time parameter for the sweep and discrete events where the calculation branches into separate sweeps on two smaller inputs, which may proceed independently.

A note to the reader. To illuminate the correspondences between different structures that appear as origami shapes we introduce some new terminology and give names to certain properties needed to track the algorithm invariants that have to be proven. In some cases we have replaced terms used in [44]. For instance, Lang uses "active reduced paths" and "active polygon" to denote features identified in the first phase of the TreeMaker algorithm. We study here the universal molecule algorithm independently of the first phase, and we want to make clear the association between the tree and the convex region that represents the "paper" to be folded. Hence we introduced the new term "Lang polygon" (instead of "active polygon") to denote *both* a convex polygon and its corresponding tree that together form a valid input for the universal molecule algorithm. Similarly, we use the more descriptive term "splitting segment" rather than "reduced active path" to denote the line segment added to the crease pattern to split the polygon at certain events during the sweep. In those instances where we have given a new name to a concept which either appears or is similar to a concept which appears in Lang's work, we have noted the name used by Lang in a footnote. We have also named certain structures after Lang, notably Lang polygons and Lang surfaces, in recognition of the fact that these concepts come primarily from his work.

Chapter outline. We start in Section 5.2 by recalling the precise definitions for all the concepts used in the chapter: metric trees, doubling polygons and surfaces, and their splitting operations. Next we introduce the specialized versions related to Lang's property, Lang polygons and Lang surfaces, and prove basic properties and relationships between them. The proof of the Main Theorem is split into three parts: *necessity*, shown in Section 5.3; the Universal Molecule algorithm and the proof of *sufficiency*, presented in Section 5.4, and *uniqueness*, shown in Section 5.5.

5.2 Lang polygons and Lang surfaces

In this section we recall from Ch. 2 the main concepts needed throughout this chapter. A *Lang polygon*[1] is an abstraction of the "piece of paper" on which an origami crease pattern will be placed, and which must satisfy certain properties, if the pattern is to be foldable into a shape resembling a tree. A piece of paper together with a crease pattern is viewed as an intrinsic surface with piecewise linear faces and zero Gaussian curvature at all interior points. A *Lang surface* is defined extrinsically, via a special placement in 3D. This is the formal concept which allows us to define, independently of any algorithm that would compute it, the special folded origami shape with a tree-like structure that Lang calls a *uniaxial base*. Both a Lang polygon and a Lang surface are derived from a metric tree T. These concepts are needed to formulate and prove the Main Theorem, and thus to fully characterize what the Universal Molecule algorithm computes.

[1] In [44] this is called an *active polygon*.

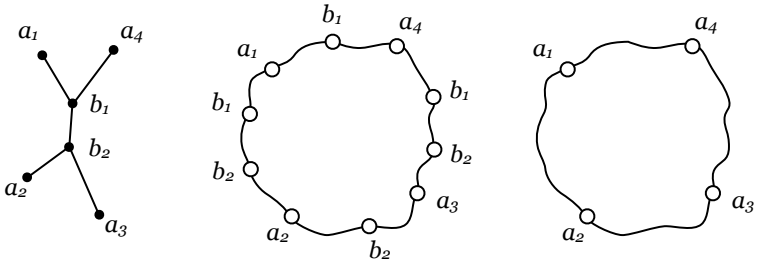


Figure 5.2: A tree (left), its doubling-cycle (center), and its leaf-cycle (right). The doubling-cycle vertices are labeled with their corresponding nodes in the tree.

5.2.1 Trees and doubling polygons

Metric trees. A ***metric tree*** (T, w) is a tree T together with a set of positive weights w defined on each arc[2] of the tree; each weight designates the ***length*** of the arc. For convenience we separate the set of nodes of T into two sets A and B, with $A = \{a_1, \ldots, a_n\}$ the leaf nodes and $B = \{b_1, \ldots, b_m\}$ the internal nodes of T. We assume that a cyclic ordering (rotation) of the incident arcs has been given for each internal node. We ***embed*** a metric tree in the xy-plane by mapping each of its nodes to a point in the xy-plane, and each arc to the line segment joining its endpoints such that the length of the line segment is equal to the length of the arc and the edges do not cross. To keep the notation simple, we drop the weights w and assume that a metric tree T has arc lengths defined from a plane embedding.

We make the tree ***kinetic*** by assigning a "speed" $s(ab)$ to each leaf arc ab. This induces a family of metric trees $T(t)$ parametrized by *time* t. Each $T(t)$ is combinatorially equivalent to T, but the length of each leaf arc ab is obtained by subtracting $t\, s(ab)$ from its original length in T. In other words, each leaf arc "shrinks" at a linear rate given by its defined speed. We call this a ***sweep*** of the kinetic tree. If we fix an embedding of T, then we give a parametrized embedding for $T(t)$ by moving each leaf

[2]To avoid possible confusions, we use *vertex* and *edge* when refering to the graph or subgraph of the UM crease pattern and *node* and *arc* for a tree.

node inwards along its incident arc at the defined speed. Fig. 5.14(left) illustrates a kinetic tree.

Doubling cycles and polygons. Given an embedded tree T, we construct its ***doubling-cycle*** by walking around T, respecting the cyclic ordering at each node and listing (with repetition) each node as it is encountered. The metric version of this concept also retains the edge lengths. The vertices of the cycle are labeled with the nodes of the tree. A ***doubling-polygon*** is an embedding of a metric doubling-cycle as a planar polygon with the given edge lengths. The ***leaf-cycle*** for a tree T is the same as the doubling-cycle except that only the leaf nodes of T are listed. See Fig. 5.2. Given a kinetic tree T, we make its doubling-cycle C_T kinetic by assigning each edge of the doubling-cycle the same shrinking speed as that of the tree. As with the tree, this gives rise to a family of doubling-cycles $C_T(t)$ parametrized by t maintaining the invariant that each $C_T(t)$ is a doubling-cycle for the tree $T(t)$. This correspondence will be used in the parallel sweep of the UM algorithm, and is illustrated in Fig. 5.14.

Notation and terminology. A *vertex* of a doubling-cycle or polygon corresponds to a *node* of a tree. **Bold face** is used to denote elements of a doubling-polygon and *italics* to denote the corresponding elements in the tree[3]. Given a tree T and a doubling-polygon P_T, $d_T(u, v)$ denotes the distance between nodes u and v in the tree and $d(\mathbf{u}, \mathbf{v})$ denotes the distance in the plane between $\mathbf{u}$ and $\mathbf{v}$. We work with convex (but not necessarily strictly convex) polygons: some vertices may have interior angle *equal to* π. Those less than π are termed ***corners*** and those equal to π are ***markers***. The chain of edges between two consecutive corners is a ***side*** of the polygon.

[3]As an example, if $\mathbf{u}$ is a vertex of a doubling-polygon, then u is its corresponding node in the tree.

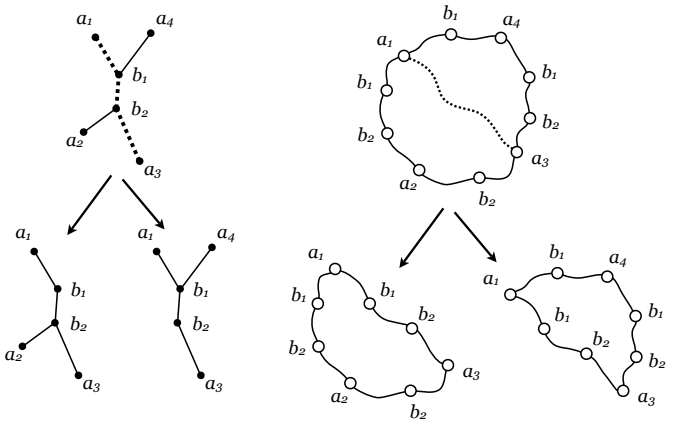


Figure 5.3: The splitting operation for a tree and its doubling-cycle. The split occurs for the pair (a_1, a_3).

Splitting. We define a ***splitting*** operation on trees and doubling-cycles. A path between two leaf nodes on the tree gives a well defined *left* and *right* side of the path. The split operation takes a tree T and two leaf nodes a_i and a_j and returns the left and right trees T_L and T_R formed by splitting T along the path $a_i \sim a_j$. See Fig. 5.3. This operation can be extended to a doubling-cycle for the tree: split the doubling cycle into two sub-chains between $\mathbf{a}_i$ and $\mathbf{a}_j$; then close each chain by gluing in a copy of the path $a_i \sim a_j$ to obtain a doubling-cycle for T_L and a doubling-cycle for T_R. We also extend this operation to doubling-polygons if the distance between $\mathbf{a}_i$ and $\mathbf{a}_j$ is equal to the length of the path $a_i \sim a_j$ in the tree. We split the polygon by adding the line segment $(\mathbf{a}_i, \mathbf{a}_j)$ to its interior and subdivide the segment so that it is metrically equivalent to the path from a_i to a_j. Figure 5.15(b) shows an example.

5.2.2 Piecewise linear surfaces

A piecewise linear metric surface is obtained by gluing together (flat and rigid) polygonal faces, along entire edges. Each edge is incident to one or two faces; in the first case it is a ***boundary edge***, otherwise it is an ***interior edge***. A vertex incident to a boundary edge is a ***boundary vertex***, otherwise an ***interior vertex***. The ***dihedral angle*** for an interior edge is the angle between the supporting planes of

its two incident faces. We assume that dihedrals are measured between 0 (inclusive) and 2π (exclusive). In this chapter we will only work with *piecewise linear metric surfaces* and hence refer to them simply as *surfaces.*

Realizations and intrinsic vs. extrinsic properties. A ***realization*** (or *isometric placement*) of a surface is given by attaching coordinates in $\mathbf{R}^3$ to each vertex of the surface such that the edges and faces maintain their size and shape; in other words, the faces behave like rigid panels. Properties of a surface which are preserved in every realization are called ***intrinsic properties***, while those which depend on the chosen realization are ***extrinsic***. For instance, the length of an edge or of a path on the surface is an intrinsic property, but the dihedral angle between two faces is extrinsic. A realization of a surface is (extrinsically) flat if all the vertices lie in one plane.

Flat surfaces. A vertex of a surface is incident to several faces, and has an angle measure on each face. Its ***angle sum*** is obtained by summing these angle measures on all incident faces. The (intrinsic, Gaussian) curvature of a piecewise linear surface at an interior vertex is defined as 2π minus the vertex angle sum. If the curvature is zero at every interior vertex, then we say that the surface is ***(intrinsically) flat***. This is "intrinsic", because a surface in $\mathbf{R}^3$ may have zero curvature everywhere, but may not be embedded in a plane. If the surface is then realized in $\mathbf{R}^3$ such that the dihedral angle of each internal edge is π, then we say that the surface is in an ***open, flat realization***. An origami crease pattern drawn on the paper but not yet folded is a surface in an open, flat realization. Once the paper is folded, the surface is still intrinsically flat, but the realization may not be extrinsically flat.

Topological disks and rings. In this chapter, we encounter only two topological classes of piecewise linear surfaces: ***disk-like surfaces***, topologically equivalent to a disk, which have a single simply connected boundary component, and ***ring-like***

surfaces, topologically equivalent to an annulus; these have two simply connected boundary components. A disk-like surface is intrinsically flat if and only if there exists an open, extrinsically flat realization of it. A ring, however, may be intrinsically flat but have no open, flat realization. An example of this is the cube with its top and bottom faces removed. Any realization of the resulting surface in the plane will have at least two edges which are "folded", i.e. have zero dihedral angle.

The boundary polygon of a disk. If we have a piecewise linear surface which is topologically a disk, and the angle sum of each of its boundary vertices is not greater than π, then we say the surface has an ***(intrinsically) convex*** boundary polygon. If the surface is also intrinsically flat, then it has a realization in the xy-plane such that none of its faces overlap. We call this the ***open, flat*** realization of the disk-like surface.

5.2.3 Lang's property and Lang polygons

Definition 5.2.1. *Given a doubling-polygon P_T for a metric tree T, we say that P_T satisfies* Lang's property *if for all pairs of vertices $\mathbf{u}, \mathbf{v}$ with corresponding tree nodes u, v their Euclidean distance is larger than their tree distance: $d_T(u, v) \leq d(\mathbf{u}, \mathbf{v})$.*

Definition 5.2.2. *Given a metric tree T with strictly positive length edges (no degeneracies), the pair (T, P_T) is said to be a* ***Lang polygon*** *for T if P_T is a convex doubling cycle for T which satisfies Lang's property.*

Fig. 5.1 illustrates these concepts. To avoid ambiguities, we also require that each vertex corresponding to a leaf node in a Lang polygon should have an interior angle strictly less than π. Using the triangle inequality we can immediately derive the following properties of Lang polygons:

Lemma 5.2.3. *Let P_T be a doubling-polygon for the tree T.*

(1) If (T, P_T) is a Lang polygon, then any vertex corresponding to an internal node is "straight" (has an interior angle of π) and thus is a "marker". Furthermore, Lang's property holds with equality for all pairs of vertices corresponding to consecutive leaf nodes.

(2) If Lang's property holds for all pairs of corners, then (T, P_T) is a Lang polygon and Lang's property holds for any pairs of vertices with at least one marker. Further, if we require that any vertex corresponding to a leaf node be a corner, then Lang's property holds with inequality for any pair of vertices where at least one is a marker.

Proof. (1) Suppose not. Let $\mathbf{b}$ be a vertex corresponding to an internal node which is not a marker. By definition of a doubling-polygon, the vertex is on a chain of edges which correspond to a path between two leaf nodes in the tree. Since the distance between the leaf nodes is equal to the length of this path and the chain of edges contains an interior vertex, namely $\mathbf{b}$, of angle not equal to π, then the endpoints in the chain are closer than the length of the path, a contradiction. The second part follows immediately. We prove (2) for a corner $\mathbf{a}$ and a marker $\mathbf{b}$. The proof is easily extended to two markers. Let $\mathbf{a}'$ and $\mathbf{a}''$ be the corners of the side containing $\mathbf{b}$. Denote the corresponding tree nodes by a, b, a', and a''. One of the paths $a \sim a'$ or $a \sim a''$ must contain b, say $a \sim a'$. Now assume for contradiction that $d_T(a, b) \geq d(\mathbf{a}, \mathbf{b})$. As a consequence of property (1), $d_T(b, a') = d(\mathbf{b}, \mathbf{a}')$. By the triangle inequality we have that $d_T(a, a') \geq d(\mathbf{a}, \mathbf{a}')$. Equality cannot hold here, since the angle $\angle \mathbf{a}\mathbf{a}'\mathbf{b}$ is (by the convexity of P_T and the fact that we disallow the vertices corresponding to leaf nodes, namely $\mathbf{a}'$, to have angle π) less than π. Thus we have $d_T(a, a') > d(\mathbf{a}, \mathbf{a}')$, a contradiction. □

Lemma 5.2.3(2) implies that if Lang's property holds for pairs of polygon corners then it holds for all pairs of vertices along the polygonal boundary.

5.2.4 Lang surfaces: overview

Our next goal is to define *Lang surfaces*. This family of surfaces in $\mathbf{R}^3$ is defined inductively from elementary "building blocks", combined using two construction operations (Sec. 5.2.6). Each surface has an associated metric tree. The preconditions for the construction operations enforce certain tree-related constraints. We also define a *surface construction tree*, which is an auxiliary structure serving as a record of the building blocks and operations used to form any particular surface. The construction tree is used to facilitate the proofs in Sec. 5.5.

Building blocks. We define two types of building block surfaces. Each one is created by embedding in the plane a kinetic tree and its kinetic doubling-cycle and extruding the kinetic doubling-cycle while the tree shrinks. The trace of the edges of the doubling-cycle during the shrinking process gives us a surface. See Fig. 5.4. Using this extrusion process we define two classes of surfaces: extrusion disks and extrusion rings. Extrusion disks are created for kinetic trees with a single internal node ("star-shaped") where the speeds are defined so that all arcs of the tree shrink to zero-length simultaneously. The extrusion is carried out until all of the tree arcs shrink and the resulting surface is topologically a disk. See Fig. 5.4(right). Extrusion rings are created for more general kinetic trees with no restriction on the speed of each arc. The extrusion process is stopped on or before any arc of the tree reaches zero-length. The resulting surfaces are topological rings. See Fig. 5.7. We fully describe the building block surfaces in Sec. 5.2.5.

Operations. We give two operations to combine surfaces. They are illustrated in Fig. 5.5. The first takes two disks and *combines* them by gluing along a boundary interval. The second takes a disk and a ring and *extends* the disk by gluing the entire boundary of the disk to one of the boundary polygons of the ring. The operations are described in detail in Sec. 5.2.6.

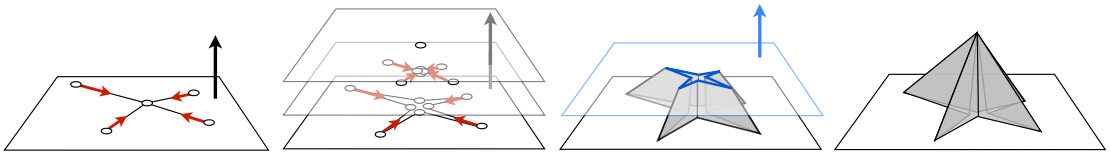


Figure 5.4: Extruding a disk. (a) The start of the extrusion process is a kinetic tree is embedded in a plane. (b) The state of the doubling-cycle at the beginning, middle, and end of the extrusion. Note that the four vertices of the doubling-cycle corresponding to the internal node of the tree overlap, but we draw them separated for visualization purposes. (c) Half-way through the extrusion process the surface is a ring. The current extrusion plane and doubling-cycle are shown in gray. (d) The final extrusion disk. The faces corresponding to the same arc of the tree overlap, but for visualization purposes we draw them slightly apart.

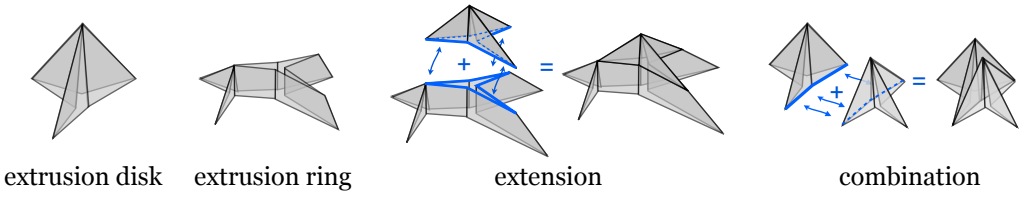


Figure 5.5: Constructing Lang surfaces. The building blocks (extrusion disks and extrusion rings) are joined using extension and combination operations.

Construction tree. Finally, in order to describe how a Lang surface is put together, we define in Sec. 5.2.7 an auxiliary structure called a *surface construction tree*. The leaves of the tree correspond to extrusion disk building blocks. Each internal node of the tree represents either a combination or an extension operation applied to its child nodes. The Lang surfaces are those surface which are constructed by this process.

5.2.5 Lang surface building blocks

There are two classes of building blocks: the first are topological disks and the second are topological rings. Both have an associated kinetic metric tree and a positive *height* value. Each surface is constructed by first placing the kinetic tree and its associated doubling-cycle in a plane. The extrusion process moves the plane upwards while moving the leaf nodes of the kinetic tree (and doubling-cycle) inwards.

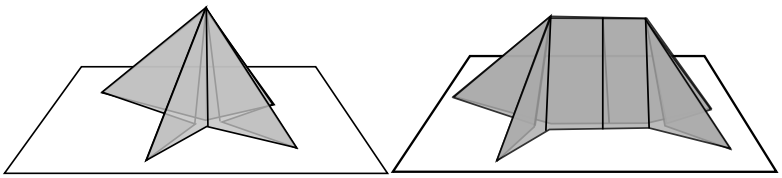


Figure 5.6: The two types of disk building blocks, corresponding to: (a) a tree with one internal node and (b) a degenerate case of a tree with two internal nodes incident to leaf arcs.

Extrusion surfaces. To form an extrusion surface for a kinetic tree we first fix an embedding of the tree in the xy-plane. We embed its doubling-cycle in the xy-plane by placing each vertex of the doubling-cycle at the same point as its corresponding tree node. We then perform the sweep of the tree and its doubling-cycle while simultaneously moving the plane containing the doubling-cycle upwards in the positive z-direction at unit speed. Note that this places all vertices corresponding to the same node of the tree vertically "on top" of one another. We parametrize the process by the *extrusion height* of the plane. The trace of the vertices and edges of the doubling-cycle form the edges and faces of a surface in $\mathbf{R}^3$ called an ***extrusion surface***. We now define two particular types of extrusion surfaces: disks and rings.

Extrusion disks. Extrusion disks (Fig. 5.4) are defined with respect to a metric tree T with a single internal node and at least three leaf nodes, and a sweep height h. We make the tree kinetic by assigning a speed of $h/d_T(a, b)$ to each arc ab. This speed has been chosen so that all arcs shrink to zero length simultaneously at $t = h$. We then perform the extrusion process detailed above until the height of the extrusion plane is h. Since the extruding doubling-polygon contracts to a single point, the resulting surface is topologically a disk. We also include in the base building blocks a degenerate situation of a tree T with two internal nodes that are incident to leaf arcs; the degeneracy arises when all leaf arcs shrink to zero at the same height h (Fig. 5.6).

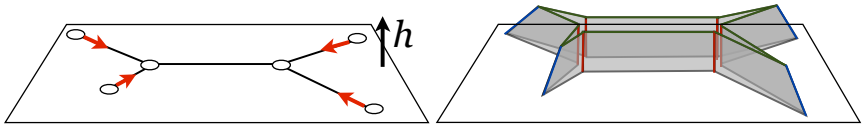


Figure 5.7: The ring building block for a tree at a height h. The faces of the surface corresponding to the same arc of the tree overlap in 3D, but, for visualization purposes, we draw them slightly apart.

To keep the presentation from becoming too technical, we handle explicitly only the first type of extrusion disk; the second, special case is a straightforward extension.

Extrusion rings. Extrusion rings are defined with respect to a kinetic metric tree T and a sweep height h. We require two preconditions: (1) no arc of the kinetic tree reaches zero length at a time $t < h$ and (2) after removing any zero-length arcs from the tree at time $t = h$, the resulting tree has at least three leaf nodes. The extrusion ring is then given by performing the extrusion process to height h. See Fig. 5.7. The resulting surface is a ring. It has two boundary components: one lies in the xy-plane and is a doubling-cycle for the initial tree T; the other lies in the $z = h$ plane and is a doubling cycle for the tree shrunken tree $T(h)$.

5.2.6 Constructing Lang surfaces

By starting with the building blocks from Sec. 5.2.5 and combining them using two operations, *combination* and *extension*, we obtain Lang surfaces.

Lang surfaces. A ***Lang surface*** S *constructed on* the tree T is a disk-like piecewise linear surface in $\mathbf{R}^3$ associated to a metric tree T whose boundary is a metric doubling-cycle for T. Lang surfaces are defined inductively. The disk building block surfaces from Sec. 5.2.5 are Lang surfaces. The ring building blocks are not Lang surfaces. Rather, they are used in conjunction with a Lang surface to form a new Lang surface. New Lang surfaces are obtained by applying the following two operations.

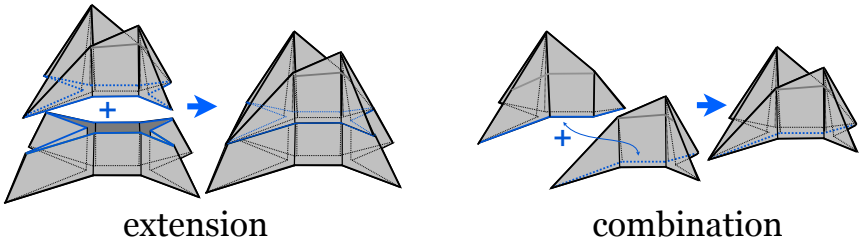


Figure 5.8: The two gluing operations for Lang surfaces. Left: the extension operation in which a surface is extended by gluing on a ring building block. Right: the combination operation in which two surfaces are glued along a border chain that corresponds to a path between consecutive leaves of the tree associated to the boundary.

The extension operation. This operation takes a Lang surface and a ring building block of height h (Sec. 5.2.5) and produces a new Lang surface S by gluing the upper boundary of the ring to the boundary of the Lang surface. We say that the input Lang surface is *extended* by the ring. The precondition for this operation is that the upper boundary of the ring and the boundary of the Lang surface be metric doubling-cycles for the same embedded metric tree. The gluing is performed by moving a Lang surface in the z-direction by h and identifying corresponding edges along the boundary. See Fig. 5.8(left). The lower boundary of the ring becomes the boundary of the surface. The tree T used to construct the ring becomes the associated tree for S. Note that when we apply the extension operation, the dihedral angle of each edge along the gluing path is π. In other words, the two faces are not "folded" along the gluing path. We could simply erase the gluing path edges and merge the two faces into a single face, but we retain the edges to aid with the proofs in Secs. 5.4 and 5.5.

The combination operation. This operation takes two Lang surfaces S_1 and S_2 and produces a new Lang surface S by gluing an interval of the boundary of S_1 to an equal length interval of the boundary of S_2. We say S is formed by *combining* S_1 and S_2. The precondition on this operation is that the corresponding trees T_1 and T_2 be related as follows. There must exist a tree T with non-consecutive leaf nodes a_i and a_j such that when T is split between a_i and a_j, T_1 and T_2 are the resulting

trees. In this case, both the boundary of S_1 and the boundary of S_2 contain a single chain of edges corresponding to the path $a_i \sim a_j$. We glue S_1 to S_2 along these chains by identifying corresponding edges. The boundary of the resulting surface S is a doubling-cycle for T. See Fig. 5.8(right).

The following definition summarizes this section:

Definition 5.2.4. *A* Lang surface *is any surface formed by starting with a collection of extrusion disks and applying extension and combination operations until a single surface results.*

5.2.7 The surface construction tree

Each Lang surface is obtained by starting with a collection of disk building blocks. The building blocks are then combined and extended using the operations from Sec. 5.2.6 until we eventually arrive at a single Lang surface. To visualize this process we associate an auxiliary ***surface construction tree*** to each Lang surface. The surface construction tree serves as a record of the building blocks and operations applied to create each surface. Each non-root node corresponds to an intermediate Lang surface from the build process and the root node corresponds to the final Lang surface. The leaves of the tree contain the disk building blocks. Each internal node represents either a combination or an extension operation applied to its child node(s). An extension operation node is labeled with input ring building block. It has a single child node representing the input Lang surface. A combination operation node is labeled with the tree T and pair (a_i, a_j) used in the combination operation applied to the Lang surfaces represented by its two child nodes. See Fig. 5.9.

Simplifying the tree. The construction tree as described above has nodes of degree 1 and 2 only. Since the combination operation always occurs along boundary chains corresponding to paths between consecutive leaf nodes, the order in which successive combination operations is applied does not matter–the end result is the same. Thus

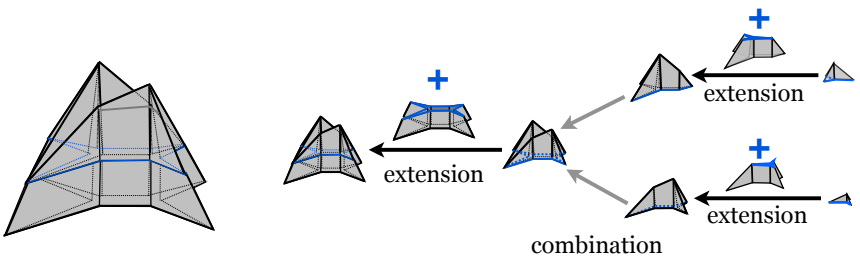


Figure 5.9: A Lang surface (left) and its construction tree (right).

we simplify the construction tree by replacing any such structure with a single parent node representing a maximal list of successive combination operations.

It is also possible to chain together multiple extension operations that can be replaced by a single extension operation. For instance suppose we have a kinetic tree T and a Lang surface S. We can create a single ring R for T of height h and then extend S using it to obtain a new surface. Or, we can create two rings R_1 and R_2, one using T of height $h/2$, and a second using $T(h/2)$ also of height $h/2$. In other words, the two rings R_1 and R_2 are equivalent to the portion of R below and above the $z = (h/2)$ plane (resp.). If we first extend S by R_2 and then extend the resulting Lang surface by R_1 we obtain exactly the same surface as before. The first method requires two nodes in the construction tree, one for S and one for the extension by R. The second requires three: one for S, one for its extension by R_2, and a third for its extension by R_1. In such cases we simplify the construction tree to use the fewest nodes possible: whenever multiple successive extension operations can be replaced by a single operation, we do so.

The extrusion for a Lang surface. Our presentation of the Lang surface construction tree proceeded in a "top-down" fashion. It begins at the building blocks, and applies operations downwards towards the root to produce the final surface. This view is useful for the proof in Sec. 5.5. However, a visually appealing view is a "bottom-up" view. We start at the root node. If the root node is an extension operation, we perform the extrusion process for its ring. If the root node is a combination operation,

then we "split" the extrusion process into separate extrusion processes for each child node. We then recursively continue, each time sweeping the extrusion plane upwards until we have covered the whole surface.

Software implementation and visualization. We refer the reader to our video [13], dedicated website (http://linkage.cs.umass.edu/origamiLang/), and online software demo for a visualization of this extrusion process. The extrusion process as defined above mirrors the sweep process in the Lang tiling: the sweeping polygons in the xy-plane and the shrinking doubling-cycles in the extrusion plane are embeddings of the same doubling-cycle. Contraction events occur between consecutive extension operations and splitting events occur at combination operations.

5.2.8 Realizations of Lang surfaces

The definition of the extrusion rings and disks given in Sec. 5.2.5 is *extrinsic*. We first fix an embedding of a kinetic tree in the xy-plane, and then extrude to obtain a surface in $\mathbf{R}^3$. This approach is visually appealing, but different embeddings of the tree lead to different surfaces in $\mathbf{R}^3$, while the underlying intrinsic surface may remain the same. It is useful to have an intrinsic description of Lang surfaces. This allows us to better analyze when changing the embedding results in the same or in different realizations, and helps with the proofs in Sec. 5.4. Towards this purpose, we give intrinsic definitions of the faces of an extrusion surface, and label their vertices with points on the metric tree.

Faces. In Sec. 5.2.5 we defined extrusion surfaces by tracing an edge of the doubling-cycle as it both moves upwards and shrinks. In the kinetic tree only the leaf nodes are moving, and thus the xy-coordinates for a vertex change only if the vertex corresponds to a leaf node. There are four types of edges for a face $f(e)$ traced by a doubling-cycle edge e: *ridge*, *perpendicular*, *base*, and *top*. The *base* edge is the initial position of e

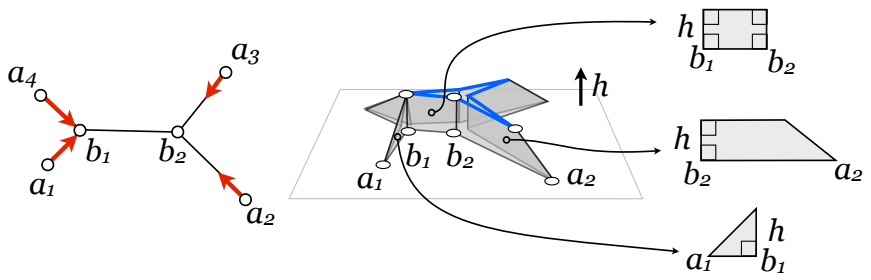


Figure 5.10: Kinetic tree (left) and its extrusion ring of height h (center). The three possible types of faces are shown (right). A face for an edge of the doubling cycle corresponding to an internal arc of the tree (b_1, b_2) extrudes to a rectangle with perpendiculars of length h. An edge corresponding to a leaf arc traces out either a right-trapezoid or right-triangle depending on if the edge shrinks to zero-length in the kinetic tree at height h.

and the *top* is the final position of e in the extrusion process. A *perpendicular* is the trace of an internal vertex and is perpendicular to the xy-plane. A *ridge* is the trace of a leaf vertex; it makes an acute angle with the base. Each face $f(e)$ is bounded by a base, two trace edges (either both perpendicular or one perpendicular and one ridge), and possibly a top edge, depending on whether e shrinks to zero-length at height h. If both endpoints of e are internal, then the traces of both endpoints are perpendiculars and the face $f(e)$ is a rectangle where both the base and top edges have length equal to that of e and the two perpendicular edges have length h. If one is a leaf node, then either e shrinks to zero-length at height h in the extrusion process or shrinks to an edge of length $||e|| - hs$ where $||e||$ denotes the length of e and s denotes the speed of the leaf node corresponding to the endpoint of e. In the first case, the traces of both endpoints meet, and so $f(e)$ is a right-triangle with e as its base edge, a perpendicular edge of length h, and a trace edge which is the hypotenuse. In the second case $f(e)$ is a right-trapezoid with base edge e, and a top edge equal to the shrunken version of e. The perpendicular between the base and the top edge has length h and the length of the ridge edge is given by elementary geometry. See Fig. 5.10. The definition above extends readily to Lang surfaces, since each face of a Lang surface is a face of one of its building block extrusion surfaces.

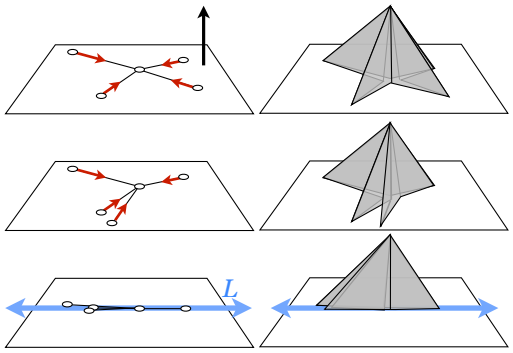


Figure 5.11: Different embeddings for the tree result in different realizations of the surface. In the bottom figure all of the tree arcs are embedded onto a common line L. The resulting realization is *uniaxial*.

Tree and height labels. We now assign two additional pieces of information ("labels") to each vertex of a Lang surface. We start with a realization of the surface with respect to a particular embedding of the tree. However, once we have defined the labels, we can forget the embedding. The first label is given by projecting the vertex orthogonally onto the xy-plane. By definition the projection of the vertex is a point on the embedded tree. We carry this back to a point on the metric tree through the inverse of the embedding–in other words, we forget the xy-coordinates of this projection point and only remember the point on the tree which is embedded there. This is the *tree label* for the vertex. The second is the *height label* which is given by the z-coordinate of the extrusion plane containing the vertex. Given this labeling the realization for some fixed embedding of the tree is given by placing each vertex at (x, y, h) where (x, y) are the coordinates of its tree label in the given embedding and h is the height label. See Fig. 5.11.

Tree projectable and uniaxial realizations. Each realization is ***tree projectable***: the projection of the surface onto the xy-plane is (geometrically) equal to the embedding of T. The boundary of the surface is mapped directly onto the embedding of the doubling cycle for T in the xy-plane, and each face is perpendicular to the xy-plane and lies in the upper half space. In a tree projectable realization, the faces which

correspond to the same arc of the tree overlap in $\mathbf{R}^3$. We can produce an "animation" of a realization by rotating the arcs of its tree around internal nodes and using the tree and height labels to lift this animation onto the surface realization. All of the faces corresponding to the same arc of the tree move as one unit around perpendicular edges[4]. Any tree T can be embedded so that all of its nodes and arcs lie on the same line. If we use such an embedding to realize a Lang surface, then all of its faces lie in a common plane and all of its boundary edges lie along a single line, or *axis*. Such a realization is called *uniaxial*. See Fig. 5.11.

5.2.9 Intrinsic curvature of Lang surfaces

The Lang surfaces constructed using the surface construction tree may have vertices of non-zero curvature. These may come from either the interior vertex of an extrusion disk building block, or from the extension and combination operations. We now investigate under what conditions the resulting surfaces are flat.

Flat building blocks. The curvature of an extrusion disk is concentrated at its one internal vertex. The curvature at that vertex is given by 2π minus its interior angle sum: $2\sum_{ab\in T} \arctan \frac{d_T(a,b)}{h}$. Given a tree, different extrusion heights result in different curvatures. However, there is always a unique positive real height for which the angle sum is 2π. Such an (intrinsically) flat extrusion disk has an open, flat realization as a convex region of the plane. An extrusion ring has no interior vertices, and so has zero-curvature trivially. As a direct consequence of the Gauss-Bonnet theorem, it can be shown that an extrusion ring has an open, flat realization if and only if its lower boundary component can be realized as a polygon in the plane such that the interior angle measure of each vertex of the planar realization is equal to the angle sum of the vertex in the surface.

[4]In origami terms, this set of faces is called a *flap* and the set of perpendiculars around which a flap rotates is sometimes referred to as a *hinge*.

Flat Lang surfaces. The combination and extension operations result in flat surfaces under two conditions: (1) the input surfaces (either a ring and a Lang surface for an extension operation, or two Lang surfaces for a combination operation) are both flat and (2) the sum of the two internal angle sums (in the two surfaces that are glued) at each vertex along the gluing path[5] is 2π.

5.3 Zero curvature Lang surfaces

We now have all the prerequisites to prove one direction of the Main Theorem: the necessity of Lang's property. Sufficiency is proven next in the Section 5.4, after describing Lang's universal molecule algorithm. Finally, we prove uniqueness in Section 5.5.

Lemma 5.3.1 (Necessity of Lang's property [44])**.** *Let S_T be a flat Lang surface constructed on a tree T and which has an intrinsically convex boundary polygon (i.e. the angle sum at each boundary vertex is less than or equal to π). Let P_{S_T} be its open, flat realization. Then P_{S_T} is a Lang polygon for T.*

Proof. Since S_T is topologically a disk, has zero curvature and has an intrinsically convex boundary, then it can be unfolded flat into the plane as a convex polygon P_{S_T}. Given two corner nodes of P_{S_T}, draw the line segment between them on the 3D surface S_T. This describes a polygonal path p where each edge traverses some face of the surface. Now fix an embedding of T and lift the surface to its corresponding tree-projectable realization in $\mathbf{R}^3$. The path p is now a path in 3-space. If we project the path down, then the projection lies on the embedding of T and covers the path in T between the leaf nodes corresponding to the corners. Thus the length of p is

[5]For the combination operation, the second requirement is always true, since the gluing path vertices that are not endpoints correspond to internal nodes of the tree, and thus are incident to perpendiculars which have angle sums of π.

not shorter than the corresponding path in the tree. This proves that P_{S_T} is a Lang polygon for T. □

Zero-curvature for Lang surfaces. The goal of origami design is to find ways of folding a flat sheet of paper into 3D shapes. In our case, the problem is to fold a convex polygonal flat sheet of paper into a Lang surface. Since the paper itself is flat and folding (in the ideal) does not introduce intrinsic curvature to the surface, whatever Lang surface we fold the paper into must be (intrinsically) flat. We now investigate some properties of such flat Lang surfaces. Our goal in the remainder of this section is to gain some intuition about what an algorithm might look like for computing a flat Lang surface from a Lang polygon by examining these properties.

Crease patterns of flat Lang surfaces. When we "unfold" a flat Lang surface S in the plane, its boundary is a Lang polygon in the plane (by Lemma 5.3.1) and the edges and faces of the Lang surface become a planar subdivision of the polygon. This subdivision is the *origami crease pattern*. Let us now look at some properties of the planar subdivisions produced by unfolding various Lang surfaces.

Unfolded extrusion disks and rings. Suppose we flatten out an intrinsically flat Lang surface and isolate one of the extrusion disks or rings used to construct the surface. Let P denote the boundary (resp. outer-boundary) of the flat realization, S denote the isolated disk (resp. ring), and h denote its extrusion height. Each ridge and perpendicular edge lie along the interior angle bisector line of the vertex of P incident to the edge (this is a trivial property of the extrusion process). Since the perpendiculars make an angle of $\pi/2$ in each of their incident faces, each vertex of P incident to a perpendicular is a marker. Similarly, since the ridges make an angle of less than $\pi/2$, each ridge is incident to a corner of P.

In an extrusion disk, we have the following trivial property which is needed for the base case of the uniqueness proof in Sec. 5.5:

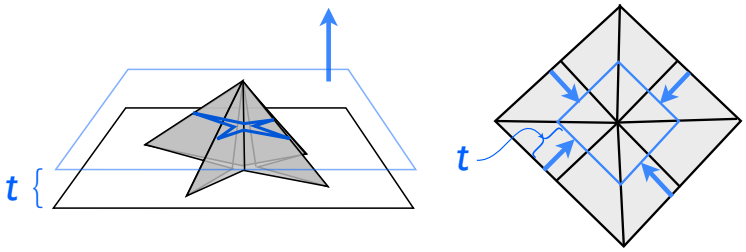


Figure 5.12: An illustration of the correspondence between the extrusion process for a flat extrusion disk (left) and a parallel sweep in its flattened out boundary polygon (right). The intersection of the $z = t$ plane with the extrusion disk (left) and the corresponding parallel offset polygon at time t (right) are shown as thick gray lines.

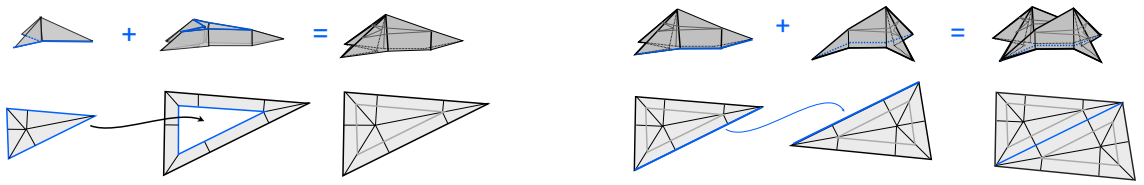
Lemma 5.3.2. *Let S be a flat extrusion disk and P_T be the Lang polygon that is the boundary of its open, flat realization. Then in the open, flat realization the interior vertex of S lies on the intersection of the angle bisectors of all vertices of P_T.*

We note the following elementary correspondence between the extrusion process used to create S and a family of *parallel offset polygons* for P. Intersect S with the $z = t$ plane for some $0 \leq t \leq h$. The result is a polygon which lies on S. Each edge of this polygon is parallel to the boundary edge of the face it lies on and is at a distance t from that edge. When we flatten out S, then the boundary polygon becomes a *parallel offset polygon*[6] defined by moving each edge of P inwards in parallel by a distance t. We can "replay" the extrusion process by the parallel offset polygons of P parametrized by t. As we increase t from 0 the sides of the polygon move inwards. We call this a *parallel sweep* of the polygon. Each vertex of the sweep traces along a ridge or perpendicular edge of S. We note the following properties. First, the parallel polygon $P(t)$ at time t is (intrinsically) equal to the the doubling-polygon in the extrusion for S at height t (see Fig. 5.12). Furthermore, since the Lang property holds when we flatten out the larger surface which contains S, it must hold for S.

[6]In [44] this is referred to as a "reduced polygon".

From this it follows that the parallel polygon $P(t)$ is a Lang polygon for the tree $T(t)$. We summarize this in the following:

Lemma 5.3.3. *Let S be an extrusion disk or extrusion ring of height h used in the construction of a flat Lang surface. Let P denote its boundary polygon in the open, flat state. Then the extrusion process constructing S of height h corresponds to a parallel sweep of P to time $t = h$ and the parallel polygon $P(t)$ at any time $0 \leq t \leq h$ is a Lang polygon for the tree $T(t)$ at height t in the extrusion process for S.*



(a) (b)

Figure 5.13: The correspondence between the 3D (top) and 2D crease pattern (bottom) for an extension operation (a) and combination operation (b) each resulting in a flat Lang surface.

Unfolding the extension operation. Suppose a flat Lang surface S is formed by extending a surface S' by a ring R. Since S has zero curvature, it follows that when we flatten out S the inner boundary of the flattened out R is the same polygon as boundary of the flattened out S'. If we independently have a flattened out crease pattern for S' and a flattened out crease pattern for R, we can produce the crease pattern for S by "pasting in" the crease pattern of S' into the inner boundary polygon of the crease pattern for R. See Fig. 5.13(a).

Unfolding the splitting operation. Suppose a flat Lang surface S_T constructed on a metric tree T is formed by a combination on two surfaces S_1 and S_2 constructed on trees T_1 and T_2 which are obtained by splitting T between leaf nodes a_i and a_j. Let p denote the gluing path in S_T. All the vertices incident to p in S_1 (resp. S_2) have

interior angle sum π, since they are incident to perpendiculars. Thus p flattens out to a straight line in the open, flat realization of S_T. See Fig. 5.13(b). Furthermore, because the boundary of S_1 is a doubling-polygon and p is a path along this boundary between consecutive leaf nodes, it follows that the length of p is equal to the distance between a_i and a_j in T. We thus have the following:

Lemma 5.3.4. *Suppose a flat Lang surface S_T is formed by a combination on two surfaces S_1 and S_2 between leaf nodes a_i and a_j. Let P_T denote the Lang polygon which is the boundary of the open, flat realization of S. Then the Lang property holds with equality for $\mathbf{a}_i$ and $\mathbf{a}_j$ in P_T, and the boundary polygons of S_1 and S_2 are given by splitting P_T between $\mathbf{a}_i$ and $\mathbf{a}_j$.*

In particular this means that if we have crease patterns for the flattened out versions of S_1 and S_2, we can construct the crease pattern for S simply by gluing the two crease patterns together along the side of each corresponding to the path between a_i and a_j.

In the next section we use these observations to describe the Universal Molecule algorithm and to prove the sufficiency of Lang's property in the Main Theorem. We also use these lemmas in the proof of uniqueness in Sec. 5.5.

5.4 Universal molecules

The proof of sufficiency for Lang's property in the Main Theorem is constructive. We start by describing Lang's universal molecule algorithm, which solves the following probch:theuniversalmolecule:lem:

Universal molecule design problem: *Given a Lang polygon (T, P_T), compute a flat Lang surface constructed on T such that its open, flat realization is P_T.*

Input and output. The *input* to the algorithm is a metric tree T and a Lang polygon P_T. The *output* is a planar graph G embedded in the plane which is the

open, flat realization of a Lang surface S_T constructed on T whose outer boundary polygon is P_T.

5.4.1 The universal molecule algorithm

We now present the universal molecule algorithm. Its pseudo-code is given below as Algorithm 1. The input (T, P_T) is a Lang polygon and the output G is a crease pattern on the interior of P_T, which is a subdivision of P_T into vertices, edges, and faces which is equivalent to the open, flat realization of a Lang surface S_T constructed on T.

Algorithm 1 UMAlgorithm(T, P_T)

```
if IsBaseCase(T, P_T) then
  return UMBaseCase(T, P_T)
end if
nextEvent ⟵ FindNextEvent(T, P_T)
(T', P'_T'), R ⟵ AdvanceSweepAndTileRing((T, P_T), nextEvent)
if nextEvent is a contraction event then
  (T', P'_T') ⟵ Contract(T', P'_T')
  G' ⟵ UMAlgorithm(T', P'_T')
else
  (T_L, P_L), (T_R, P_R) ⟵ Split((T', P'_T'), nextEvent.(a_i, a_j))
  G_L ⟵ UMAlgorithm(T_L, P_L)
  G_R ⟵ UMAlgorithm(T_R, P_R)
  G' ⟵ MergeCreasePatterns(G_L, G_R)
end if
G ⟵ MergeCreasePatternWithRing(G', R)
return G
```

Recursive procedure: the sweep. To compute the crease pattern the algorithm first performs a sweep in the tree (in which its leaf arcs shrink) and in the polygon (in which its sides move inwards and the vertices move along their respective angle bisectors) for some time interval $[0, t]$. See Fig. 5.14. We think of each recursive call as "resetting the clock" for this sweep: the input (T, P_T) represents the state of its own "local" sweep starting at time 0. The sweep is then performed to an event time

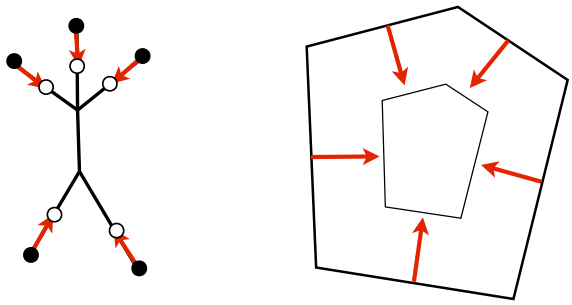
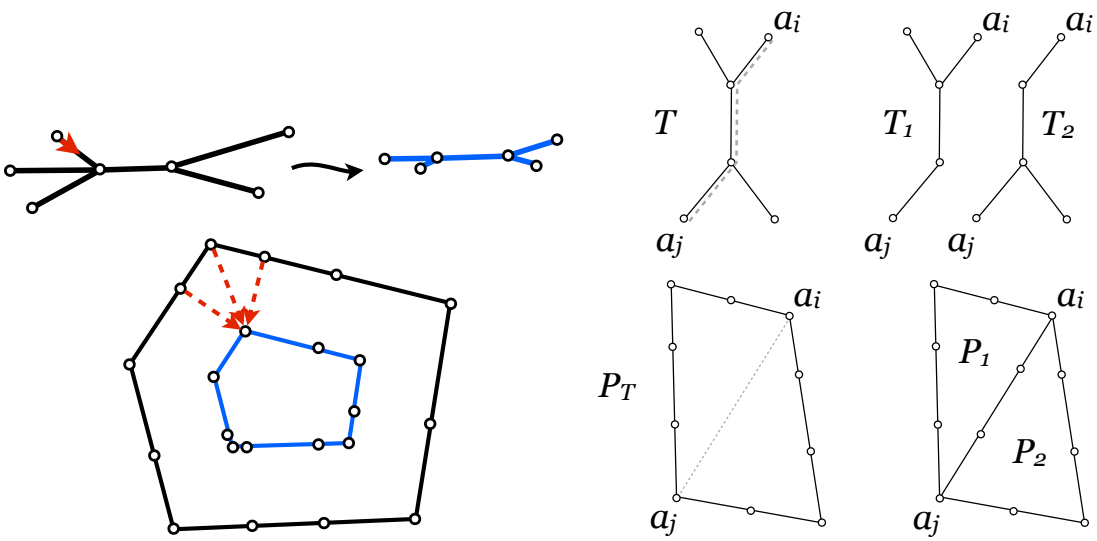


Figure 5.14: The sweeping process in a tree and the parallel sweep in its corresponding Lang polygon.

t satisfying certain properties defined shortly. Throughout the sweep we maintain the invariant that the polygon remains a Lang polygon for the shrinking tree such that the Lang property holds with strict inequality for any pair of non-consecutive corners. The event time t is the smallest time that this property is violated. Let T' and $P'_{T'}$ be the sweeping tree and polygon at time t. The trace of the vertices and edges of the sweeping polygon defines a tiling of an annular region, called a *ring tiling*, with outer boundary P_T and inner boundary $P'_{T'}$. The trace of each edge defines a face and the trace of each vertex defines an edge of this region. In the pseudo-code, the FindNextEvent subroutine computes the time t at which the sweep stops and returns a structure *nextEvent* which stores t, the event's type, and any additional information required to process the event. The AdvanceSweepAndTileRing subroutine returns the tree T' and polygon P'_T at time t in the sweep and an embedded planar map R which is the ring tiling between P_T and P'_T. Note that it is possible that the event time $t = 0$. In this case AdvanceSweepAndTileRing simply returns T and P_T and the ring tiling R is just the polygon P_T.

Events. As we perform the sweep of the tree and polygon in each recursive call, we maintain the properties that the polygon remains a Lang polygon for the tree and the Lang property holds with strict inequality for all pairs of non-consecutive corners. The event time t is the smallest time at which this property is violated. One of three



(a) A contraction event.

(b) A splitting event.

Figure 5.15: The two event types encountered by the sweep. At the contraction event in (a) a leaf arc of the tree (denoted by the dark gray arrow) and its corresponding edges in the Lang polygon contract in the sweeping tree and polygon. At a splitting event (b), the distance between two non-consecutive corners $\mathbf{a}_i$ and $\mathbf{a}_j$ in the polygon is equal to the distance in the tree between a_i and a_j. The tree is split along the path between the two nodes and the polygon is split by introducing a segment between the two corners and subdividing it so that it is equivalent to the splitting path in the tree. The sweep continues in the pair (T_1, P_1) and the pair (T_2, P_2).

things may occur: either a base case occurs, in which the entire polygon and tree shrink to a single vertex; or some edges of the polygon and arcs of the tree shrink to zero length, leading to a *contraction event*; or the Lang property fails to hold with strict inequality. In the third case, at the time t of failure, the length of the path for some pair of non-consecutive leaf vertices (a_i, a_j) is equal to the distance between the corresponding corners $(\mathbf{a}_i, \mathbf{a}_j)$. We call this last event a *splitting event* and the pair a *splitting pair*[7], because the algorithm restores Lang's property on the tree T' and doubling-polygon $P'_{T'}$ by splitting them in two along a segment, resp. a path corresponding to the pair of leaf nodes (a_i, a_j). See Fig. 5.15.

Detecting events. In order to properly discuss the events described above, we need to define the shrinking speed for each leaf arc in the tree. Given an arc a, its speed is defined with respect to the angle $\theta_{\mathbf{a}}$ of its corresponding corner in the polygon. We shrink the leaf arc incident a at a rate of $1/\tan(\theta_{\mathbf{a}}/2)$. The length of any internal arc is held constant. This maintains the property that each arc shrinks at exactly the same rate as its corresponding edges in the polygon (see Fig. 5.16). Thus if an arc of the tree shrinks to zero-length, both of its corresponding edges in the polygon shrink to zero-length, and vice versa. To detect events, we need to check two things. First we need to find the smallest time t at which some leaf arc and its corresponding edges shrink to zero-length. This is done by solving for all leaf nodes a the time t at which $t/\tan(\theta_{\mathbf{a}}/2)$ is equal to the length of its incident arc. Second, we find the time of the next splitting event by finding the pair of non-consecutive leaf nodes (a_i, a_j) that minimize t in the following equation:

$$||(\mathbf{a}_j + t\,\vec{\text{bis}}(\mathbf{a}_j)) - (\mathbf{a}_i + t\,\vec{\text{bis}}(\mathbf{a}_i))|| = d_T(a_i, a_j) - t(1/\tan(\theta_{\mathbf{a}_i}/2) + 1/\tan(\theta_{\mathbf{a}_j}/2))$$

[7] In [44] this is referred to as an "active reduced path".

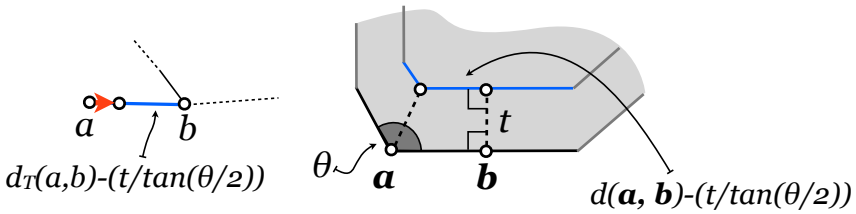


Figure 5.16: The sweep for an edge of the polygon (right) and its corresponding arc of the tree (left). The sweep maintains the property that the length of an edge of the polygon is the same as the length of its corresponding arc in the tree.

(where $\theta_{\mathbf{a}_i}$ denotes the interior angle and $\vec{\text{bis}}(\mathbf{a}_i)$ denote the interior angle bisector scaled to length $1/\sin(\theta_{\mathbf{a}_i}/2)$ for the corner $\mathbf{a}_i$). The left side of the equation is the distance between corners $\mathbf{a}_i$ and $\mathbf{a}_j$ in the sweeping polygon $P_T(t)$ at time t, and the right side is the distance in the tree between nodes a_i and a_j in the shrinking tree $T(t)$. In the pseudo-code this step is handled by the FindNextEvent procedure. The *nextEvent* data structure stores the time t of the next event, and if the next event is a splitting event it also stores the splitting pair $(\mathbf{a}_i, \mathbf{a}_j)$.

Processing events. At a contraction event, a leaf arc of the tree and its two corresponding edges in the polygon have shrunk to zero length. To process this we remove the arc from T' and contract the two edges of $P'_{T'}$ (replacing them with a single vertex). In the pseudo-code this is handled by the Contract procedure which returns the resulting pair (T', P'_T). We then recursively invoke the algorithm on (T', P'_T) to obtain a crease pattern G' for the interior of P'_T. Finally, we invoke the subroutine MergeCreasePatternWithRing to merge the resulting crease pattern G' with the ring tiling R by replacing the inner boundary polygon of R with G' and returns the resulting crease pattern G.

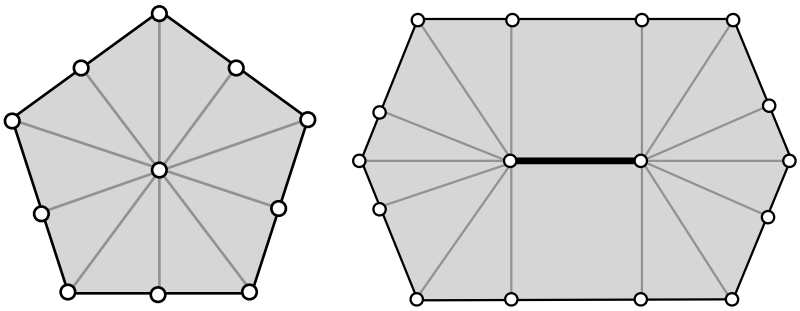
At a splitting event for a splitting pair (a_i, a_j), we split the tree between a_i and a_j, and split the polygon between $\mathbf{a}_i$ and $\mathbf{a}_j$ using the splitting operations described in Sec. 5.2. This results in two pairs: (T_L, P_L) and (T_R, P_R) (the left and right trees paired with the left and right polygons resulting from the splitting operations on trees

and doubling-polygons). In the pseudo-code this is handled by the SPLIT operation. We then recursively invoke the algorithm on both split polygons to obtain crease patterns on the interiors of P_L and P_R. We next invoke the subroutine MERGESPLITCREASEPATTERNS to merge these into a crease pattern G' for the polygon P'_T. Finally we merge G' with R using MERGECREASEPATTERNWITHRING as with the contraction events and return the resulting crease pattern G.

We note here that if we assume the correctness of the algorithm for the recursive calls, then the MERGECREASEPATTERNWITHRING subroutine is the open, flat version of the extension operation for Lang surfaces (Fig. 5.13(a)). Similarly, the MERGESPLITCREASEPATTERNS subroutine is the open, flat version of the combination operation for Lang surfaces (Fig. 5.13(b)).

Simulatenous events. It is possible for multiple events to occur simultaneously. In that case we first process all contraction events in arbitrary order, and then process any one of the splitting events. For the proof of sufficiency of the Main Theorem which we give in the next section, we do not need to worry about whether this arbitrary order may result in different crease patterns. However, in order to prove uniqueness we show in Sec. 5.5 that the order in which we process such simultaneous events does not matter and regardless of order we obtain the same final output crease pattern.

Base cases. A Lang polygon (T, P_T) is a base case if its next event t is a contraction event where all of the edges/arcs shrink to zero-length simultaneously. The trace of the parallel sweep from P_T to $P'_{T'}$ traces out a disk with P_T as its outer-boundary. It is necessarily the case that the point p at which all edges of $P'_{T'}$ contract lies on the interior of P_T and on the angle bisector lines for each vertex of P_T (including the markers). In the pseudocode this property is checked by the ISBASECASE subroutine call. If it returns true, then the algorithm invokes the UMBASECASE subroutine which returns a crease pattern for P_T. The subroutine works by finding the point p at the



(a) (b)

Figure 5.17: The base cases, in which the sweeping polygon contracts to: (a) a single point and (b) (the degenerate case) a segment.

intersection of all of the bisectors and subdividing the interior of P_T by adding edges from each vertex of P_T to p. It is also possible to have a degenerate case (Fig. 5.17), when the contracted polygon $P'_{T'}$ has only two sides (i.e. the polygon is a double covered line segment). The treatment of this case is a straightforward extension of the generic situation; to keep the presentation uncluttered[8], we omit the details.

Complexity. Each time the algorithm is recursively invoked, it is passed an input tree with strictly fewer leaves than the parent invocation (but at least three). Eventually we reach a base case, the simplest of which being when the input tree T has exactly one internal node. In this case it can be shown by elementary geometry that the angle bisectors of all corners and markers in the polygon P_T intersect at a common point. Thus the total number of events processed by the algorithm is $O(n)$. The direct implementation runs in $O(n^3)$: at each recursive call the algorithm finds the next event by first checking (in $O(n)$ time) when each edge contracts and then it checks when a splitting event will occur, for each of the $O(n^2)$ pairs of non-consecutive corners. This has recently been improved to $O(n^2 \log n)$ by using better data structures [14]. Consequently we have:

[8]This degenerate base case corresponds to the degenerate extrusion disks discussed in Sec. 5.2.5.

Corollary 5.4.1. *The universal molecule algorithm always terminates.*

5.4.2 Proof of Main Theorem: Sufficiency

We now prove the sufficiency claim of the Main Theorem by showing that the algorithm described in Sec. 5.4.1 is correct: for a Lang polygon (T, P_T) as input, the planar subdivision returned by the algorithm is the open, flat realization of some Lang surface S_T constructed on T which is isometric to P_T. The proof is divided into two parts. The first part proves the invariant property of the input: that each time we invoke the algorithm on a tree T and polygon P_T, the input is *valid*, i.e. (T, P_T) is a Lang polygon. The second part proves the invariant property of the output: given a Lang polygon (T, P_T), the algorithm returns a crease pattern G which "fills in" the interior of P_T and which is the open, flat realization of a Lang surface S_T constructed on T with boundary P_T. In both cases the proof is inductive.

Lemma 5.4.2. *If the* UMAlgorithm *is invoked on a Lang polygon (T, P_T) then, when it makes a recursive call it passes as input a valid Lang polygon.*

Proof. The algorithm first finds the time t of the next event (which may be equal to 0) and computes a tree T' and polygon $P'_{T'}$ by advancing the sweep in T and P_T. We first show that T' is a doubling-polygon for $P'_{T'}$. At this point the algorithm has not yet performed any contraction operations on the tree and polygon, so T' and T, resp. $P'_{T'}$ and P_T are combinatorially identical. The claim then follows directly from the fact that we defined the speed at which each leaf arc shrinks to match the exact speed at which its corresponding edge in the sweeping polygon shrinks.

We next prove, by contradiction, that for any pair of corners $(\mathbf{a}_i, \mathbf{a}_j)$ the Lang property holds (possibly with equality). Assume not, and let $\mathbf{a}_i$ and $\mathbf{a}_j$ be a pair for which the Lang property does not hold at time t. But Lang's property does hold for the pair of vertices $(\mathbf{a}_i, \mathbf{a}_j)$ in the input (T, P_T). Since the distances between vertices in $P'_{T'}$ and leaf nodes in T' change continuously over the sweep interval, this implies

that at some time $0 \leq t' < t$, the Lang property must hold with equality for $(\mathbf{a}_i, \mathbf{a}_j)$. But this contradicts the fact that t is the time of the first event in the sweep.

Next the algorithm contracts any zero-length arcs from the tree T' and the corresponding edges from $P'_{T'}$. This operation restores the property that $P'_{T'}$ is non-degenerate and trivially maintains convexity. It also does not change any distances in the tree or the polygon, and so by continuity, the Lang property holds (possibly with equality).

Finally, if we are at a splitting event, the algorithm splits the tree T' and polygon $P'_{T'}$ using the splitting operation defined in Sec. 5.2. Since this operation does not change distances, the resulting pairs (T_L, P_L) and (T_R, P_R) are trivially Lang polygons. □

The next two lemmas prove, by induction, the invariant properties of the output. Together they imply the correctness of the Universal Molecule algorithm. The first lemma handles the inductive step and the second the base case.

Lemma 5.4.3. *Given a Lang polygon (T, P_T), if each recursive call correctly computes a Lang surface for its input polygon, then the algorithm correctly computes a crease pattern which is the open, flat realization of a Lang surface constructed on T with boundary P_T.*

Proof. Let G be the crease pattern computed by the algorithm. We are going to show that G is the open, flat realization of a Lang surface S_T constructed on T. Let $(T', P'_{T'})$ be the Lang polygon at the next event and t be the time of the next event. The algorithm first computes the ring tiling R between P_T and $P'_{T'}$. It is trivial to show that R is equivalent to an extrusion ring R_T for T of height t (where the speed for each leaf node in T is defined to be the same as in the sweep used to create R), see Sec. 5.3.

To complete the proof, we first show that after processing either type of event we are left with a crease pattern G' whose boundary is $P'_{T'}$ and G' is the open, flat

realization of a Lang surface $S'_{T'}$ constructed on T'. The result then follows by the fact that when G' is merged with R the resulting crease pattern G is trivially the open, flat realization of the Lang surface created by extending $S'_{T'}$ by the ring R_T.

It remains to show that such an $S'_{T'}$ exists. If we are not at a splitting event, then this follows directly from the inductive hypothesis. If we are at a splitting event, then the algorithm splits $(T', P'_{T'})$ using some splitting pair $(\mathbf{a}_i, \mathbf{a}_j)$ producing two Lang polygons (T_L, P_L) and (T_R, P_R). The two polygons are incident along the added splitting segment between $\mathbf{a}_i$ and $\mathbf{a}_j$. By inductive hypothesis there exists a Lang surface S_L (resp. a S_R) constructed on T_L (resp. T_R) such that the open, flat realization of S_L (resp. S_R) is G_L (resp. S_L). S_L and S_R meet the preconditions for the combination operation along the chain of boundary edges which "flatten out" to the splitting edge between $\mathbf{a}_i$ and $\mathbf{a}_j$. Let $S'_{T'}$ be the resulting surface. The open, flat realization of $S'_{T'}$ is equivalent to the crease pattern G' produced by merging G_L and G_R. □

Lemma 5.4.4. *Let (T, P_T) be a base case for the universal molecule algorithm and G be the returned crease pattern. Then there exists a Lang surface S_T constructed on T for which G is an open, flat realization.*

Proof. By definition in the sweep of P_T, all edges contract simultaneously at some time t. Each face of the resulting crease pattern G is a right triangle with one leg equal to an edge of P_T, one leg equal to the trace of the incident marker with length t and a hypotenuse equal to the trace of the incident corner. Let S_T be the extrusion disk of height t for T. It can be verified mechanically from the definition of the extrusion disk that G is the open, flat realization of S_T. □

We summarize as:

Corollary 5.4.5 (Correctness of the Universal Molecule algorithm). *Given a Lang polygon (T, P_T) the universal molecule algorithm computes a crease pattern G on P_T which is the open, flat realization of a Lang surface S_T constructed on T.*

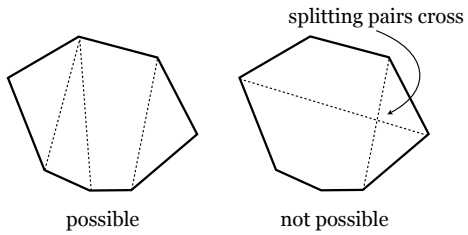


Figure 5.18: The dotted lines denote pairs of corners satisfying Lang's property with equality. (Left) None of the pairs cross. (Right) Two crossing pairs: an impossibility in a Lang polygon, due to Lemma 5.5.1.

This completes the proof of the sufficiency of the Lang property in the statement of the Main Theorem.

Corollary 5.4.6 (Main Theorem: Sufficiency). *If (T, P_T) is a Lang polygon then there exists a Lang surface S_T constructed on T for which the boundary of the open, flat realization of S_T is P_T.*

We now turn to proving the uniqueness claim in the Main Theorem.

5.5 Uniqueness

We complete the proof of the Main Theorem by showing the uniqueness of Lang's surface constructed on T and isometric to P_T. We first prove that the output of the Universal Molecule algorithm is unique: no matter what order we process simultaneous events in, the resulting crease pattern is the same. We then use this fact to prove the uniqueness claim of the Main Theorem by showing that if S_T is a zero-curvature Lang surface constructed on T which flattens out to a Lang polygon P_T for T, then the crease pattern returned by the Universal Molecule algorithm for (T, P_T) coincides with the open, flat realization of S_T.

Uniqueness of universal molecules. We show that the order in which we remove zero-length arcs/edges from the tree and polygon at a contraction event in the Universal Molecule algorithm does not effect the final outcome. We first show that it is

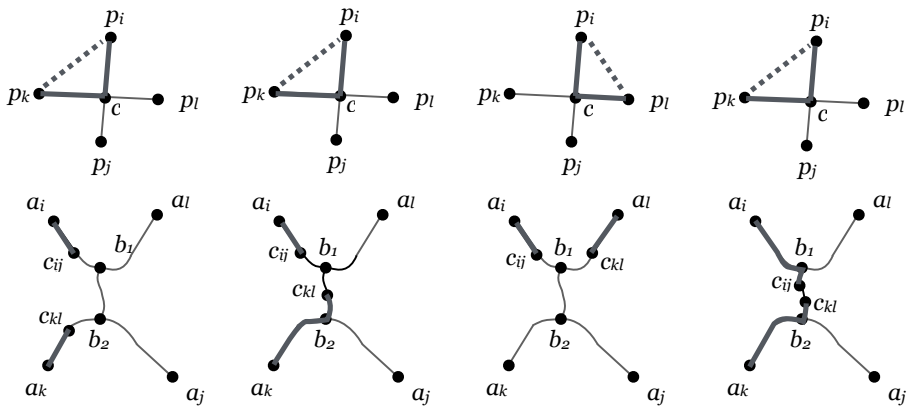


Figure 5.19: The contradiction for Lemma 5.5.1. (Top) Two crossing pairs. (Bottom) The part of the tree corresponding to the crossing pairs. Illustrated are the cases where the crossing point is on the path from a_i to b_2 in the tree. The cases where it lies on a_j to b_2 are symmetric. The dotted line depicts the contradiction where Lang's property does not hold.

not possible to have two simultaneous and *distinct* splitting events. We say that two pairs of vertices $(\mathbf{a}_i, \mathbf{a}_j)$ and $(\mathbf{a}_k, \mathbf{a}_l)$ *cross* (Fig. 5.18) if, in a counter-clockwise walk of the polygon, the vertices are encountered in the order $\mathbf{a}_i$, $\mathbf{a}_k$, $\mathbf{a}_j$, $\mathbf{a}_l$. If two crossing pairs give rise to two simultaneous splitting events, then we may obtain distinct Lang surfaces, by splitting the polygon along the two separate pairs. In the case when we split along the $(\mathbf{a}_i, \mathbf{a}_j)$ pair, the other vertices $\mathbf{a}_k$ and $\mathbf{a}_l$ would lie, after the split, in different polygons. Thus they won't lead to any further splitting event. Similarly for the $(\mathbf{a}_k, \mathbf{a}_l)$ pair.

The following lemma shows that this situation cannot occur. For completeness, we include its proof, which appeared previously in [31].

Lemma 5.5.1 ([31, Lemma 16.4.2])**.** *Let* $(\mathbf{a}_i, \mathbf{a}_j)$ *and* $(\mathbf{a}_k, \mathbf{a}_l)$ *be two pairs of vertices of a Lang polygon* (P_T, T) *such that Lang's property holds with equality for each pair. Then* $(\mathbf{a}_i, \mathbf{a}_j)$ *and* $(\mathbf{a}_k, \mathbf{a}_l)$ *do not cross.*

Proof. For a contradiction, we assume that there exist such pairs $(\mathbf{a}_i, \mathbf{a}_j)$ and $(\mathbf{a}_k, \mathbf{a}_l)$ which cross at a point c. The possible cases are illustrated in Fig. 5.19.

We look at the tree formed by the union of the two paths $a_i \sim a_j$, $a_k \sim a_l$. Either the two paths cross at a node b, or they overlap over some sub-path $b_1 \sim b_2$. In either case we will arrive at a contradiction.

In the plane, the crossing point c is at a distance d_1 from p_i and d_2 from p_k. In the tree, let c_{ij} be the point along the path $a_i \sim a_j$ which is d_1 away from a_i. Similarly, let c_{kl} be the point along the path $a_k \sim a_l$ which is d_2 away from a_i. There are several ways c_{ij} and c_{kl} can be placed. The three leftmost illustrations of Fig. 5.19 show the three possibilities where c_{ij} lies on the $a_i \sim b_1$. The fourth shows the case when both c_{ij} and c_{kl} lie on $b_1 \sim b_2$. All other configurations are symmetric to these.

We now analyze the "quadrants" of the tree given by consecutive pairs of leaves: (i,k), (j,k), (j,l), (i,l). Each "quadrant" (m,n) (where $m \in \{i,j\}$ and $n \in \{k,l\}$) corresponds to the path $a_m \sim a_n$ in the tree, and to the triangle $p_m p_n c$ in the plane. In any placement of c_{ij} and c_{kl}, one of the quadrants (m,n) is such that the path $a_m \sim a_n$ contains c_{ij} and c_{kl} in that order (i.e. $a_m \sim a_n = a_m \sim c_{ij} \sim c_{kl} \sim a_n$). By triangle inequality we have $d(p_m, p_n) < d(p_m, c) + d(p_n, c)$, with the righthand term equal to $d_T(a_m, c_{ij}) + d_T(a_n, c_{hk})$. This contradicts Lang's property. Fig. 5.19 illustrates the contradiction in each case. □

The previous lemma allows us to safely split a Lang polygon, at simultaneous splitting events, in an arbitrary order. After applying a split, all of the remaining splitting pairs are still intact, in the sense that both vertices of a remaining splitting pair belong to the same split polygon. The recursive call on the split polygons will find that the next event is at time $t = 0$ and will arbitrarily select one of the remaining splitting pairs to split on; the end result is the same. Given $k+1$ splitting pairs, we obtain the same k polygons after recursively applying the split operation in any arbitrary order.

Next, we need to show that when Lang's property holds with equality, we have to stop the sweep. If we could continue past that point (where Lang's property held with equality), then we could potentially use this fact to generate different Lang surfaces,

depending on whether we decided to split at this point or continue the sweep. We prove that the sweep must stop by showing that if we were to continue the sweep past a splitting event, the polygon would no longer be a Lang polygon, and thus no Lang surface could include the ring traced out by the sweep. This follows from:

Lemma 5.5.2. *Let (T, P_T) be a Lang polygon, on which we perform a universal molecule sweep up to the first contraction event and ignoring the possible splitting events. Then, for any pair of non-consecutive corner nodes, their distance in the plane decreases at a strictly faster rate than the corresponding distances in the tree.*

Proof. Let $\mathbf{a}_i$ and $\mathbf{a}_j$ be two non-consecutive corners of P_T. Let $\vec{\text{bis}}(\mathbf{a}_i)$ and $\vec{\text{bis}}(\mathbf{a}_j)$ be the velocity vectors of $\mathbf{a}_i$ and $\mathbf{a}_j$ in the sweep. Recall that the magnitudes of these vectors are $1/\sin(\theta_{\mathbf{a}_i}/2)$ and $1/\sin(\theta_{\mathbf{a}_j}/2)$ resp. The instantaneous rate of change between $\mathbf{a}_i$ and $\mathbf{a}_j$ can be found by the following construction: project $\mathbf{a}_i$ and $\mathbf{a}_j$ down onto the line L containing $\mathbf{a}_i$ and $\mathbf{a}_j$. The lengths of the projected vectors are (by elementary trigonometry) $\cos(\alpha_i)/\sin(\theta_{\mathbf{a}_i}/2)$ and $\cos(\alpha_j)/\sin(\theta_{\mathbf{a}_j}/2)$ where α_i and α_j denote the angles between L and $\vec{\text{bis}}(\mathbf{a}_i)$ and $\vec{\text{bis}}(\mathbf{a}_j)$ resp. By convexity $\alpha_i < \theta_{\mathbf{a}_i}/2 < \pi/2$ and $\alpha_j < \theta_{\mathbf{a}_j}/2 < \pi/2$. Therefore $\cos(\alpha_i)/\sin(\theta_{\mathbf{a}_i}/2) + \cos(\alpha_j)/\sin(\theta_{\mathbf{a}_j}/2) > \cos(\theta_{\mathbf{a}_i}/2)/\sin(\theta_{\mathbf{a}_i}/2) + \cos(\theta_{\mathbf{a}_j}/2)/\sin(\theta_{\mathbf{a}_j}/2) = 1/\tan(\theta_{\mathbf{a}_i}/2) + 1/\tan(\theta_{\mathbf{a}_j}/2)$ which is the rate at which the distances between the leaf nodes decreases, proving the lemma. □

Corollary 5.5.3. *Let (T, P_T) be a Lang polygon on which the first event in the Universal Molecule algorithm is a splitting event at time t. Then, if we continue the sweep (without splitting) by any sufficiently small Δt past the event, then the sweeping polygon stops being a Lang polygon for the shrinking tree.*

Proof. Since the distances in both the shrinking tree and sweeping polygon are equal at the event, and the distances in the polygon are decreasing at a faster rate than the

distances in the tree (by Lemma 5.5.2), then the corollary follows by the fact that pairwise distances change continuously in both the tree and polygon. □

We obtain:

Corollary 5.5.4. *The output G of the Universal Molecule algorithm on a Lang polygon (T, P_T) is well defined and unique.*

Uniqueness of the Lang surface S_T. We prove:

Lemma 5.5.5 (Uniqueness)**.** *For any Lang polygon (T, P_T), there exists a unique Lang surface S_T constructed on T and isometric to P_T.*

Shortly, we say that S_T flattens out to (T, P_T). To prove this we show that if we are given a flat Lang surface S_T that flattens out to a Lang polygon (T, P_T), then if we run the Universal Molecule algorithm on (T, P_T) the returned crease pattern G is the open, flat realization of S_T. The uniqueness then follows from the uniqueness of the output of the algorithm (Cor. 5.5.4). In particular this proves that all flat Lang surfaces are producible by the Universal Molecule algorithm.

Lemma 5.5.6. *Let T be a metric tree and P_T be a Lang polygon for T. Let S be a (zero curvature) Lang surface constructed on T and isometric to P_T. Then the output G of the Universal Molecule algorithm on (T, P_T) is the open, flat realization of S.*

Proof. The proof is by induction on the construction tree.

Base case. The construction tree for S is, in this case, a single node and S is an extrusion disk. By Lemma 5.3.2 its single internal vertex $\mathbf{v}$ lies on the (intrinsic) angle bisectors of each boundary vertex. We now show that P_T has to be a base case for the Universal Molecule algorithm, hence the result of algorithm is S.

Since all angle bisectors of P_T intersect at a common point p, it follows that the next possible contraction event is a contraction of all edges/arcs simultaneously. However, we still need to show that no splitting event can occur in the sweep before we

reach this point. Indeed, if we assume that a splitting event occurs, then Lemma 5.5.2 can be applied. Since the distances satisfy Lang's property initially, they change continuously over the course of the sweep, and trivially become equal at the contraction event, then it is not possible that they become equal at any other time during the sweep. Thus a contradiction.

For the inductive step, we assume that the lemma is true for the Lang surfaces corresponding to the subtrees for the children of the root node of the construction tree for S. Then we show the lemma is true for S. There are two cases, depending on whether the root node of the construction tree for S corresponds to an extension or to a combination operation.

Case 1 (Extension): S is formed by an extension operation on an extrusion ring R of height h and a Lang surface S'. The faces from R lie between the $z = 0$ and $z = h$ planes and the faces of S' all lie above the $z = h$ plane. Suppose we run the algorithm on (T, P_T) and the first event is at time t. We need to show that the time of the event in the parallel sweep is equal to the extrusion height for the ring (i.e. $t = h$). By Lemma 5.3.3 it follows that for any $t' \leq \min\{t, h\}$, the sweep polygon at t' is equivalent to the intersection of R with the $z = t'$ plane.

We next claim that $t \geq h$. Assume not. If t is a contraction event, then some edge of P_T shrinks to zero length at time t. Then the same edge shrinks at height $t < h$ in the extrusion process for R contradicting the fact that the extrusion process stops if such an event occurs. If t is a splitting event for some pair of non-consecutive corner nodes $(\mathbf{a}_i, \mathbf{a}_j)$, then Lang's property holds with equality: the length of the line segment s between $\mathbf{a}_i$ and $\mathbf{a}_j$ equals the tree distance. Both vertices lift onto R into the $z = t$ plane. Then s lifts onto S to some polygonal path in 3D. Since the intersection of R with the $z = t$ plane is geometrically equivalent to the tree at time t', the length of the projection of this path onto the $z = t'$ plane must be greater than or equal to the length of the corresponding distance in the tree at time t'. But this

means that the entire path must lie in the $z = t'$ plane and thus along the border of the sweep polygon at time t', a contradiction.

We now show that $t = h$. The surface S' was either formed by an extension or a combination operation. If an extension, then it must be the case that the tree associated to S' has fewer arcs than T. Otherwise it violates the simplicity requirement that no consecutive extension operations can be replaced by a single operation (Sec. 5.2.7). Therefore at height h an edge in the extrusion process shrinks to zero-length. By Lemma 5.3.3 a contraction event occurs in the algorithm at time $t = h$. Now suppose S' is formed by a combination operation. Then by Lemma 5.3.4, a splitting event occurs at time $t = h$.

We now have that the trace of the parallel sweep to the first event is equivalent to R and thus the polygon $P'_{T'}$ at time t is equivalent to the boundary of S'. By inductive hypothesis when we recursively run the algorithm $T', P'_{T'}$ it returns S'. Thus the output of the algorithm on (T, P_T) is S.

Case 2 (Combination): Let S_1 and S_2 be the surfaces combined to form S by gluing along a path between boundary vertices $\mathbf{a}_i$ and $\mathbf{a}_j$. We showed above, when analyzing Case 1, that the combination operation implies that Lang's property holds with equality for the pair $(\mathbf{a}_i, \mathbf{a}_j)$ in P_T. If we run the Universal Molecule algorithm on (T, P_T), then the first event is a splitting event at time $t = 0$ between $(\mathbf{a}_i, \mathbf{a}_j)$ producing (T_1, P_1) and (T_2, P_2). It follows from the definition of the splitting and combination operations that the associated tree and boundary polygon for S_1 is (T_1, P_1) (similarly for S_2). By the inductive hypothesis, when the algorithm recurses on (T_1, P_1) and (T_2, P_2) then it returns S_1 and S_2. Therefore, the output of the algorithm on (T, P_T) coincides with S. □

This concludes the proof of the Main Theorem.

5.6 Conclusion

The Universal Molecule algorithm first appeared in Robert Lang's paper presented at the Symposium on Computational Geometry in 1996, as part of his TreeMaker method for origami design [44]. Since then, various descriptions have been available [45, 46, 31], but they all stopped short of giving complete details and proofs connecting the input and output of the algorithm with the desired 3D shape. Even in the most expanded account [31] it is stated that correctness "requires a careful analysis of the details, which we do not attempt here." To date no such analysis has been given, leaving a general impression that the details are too formidable. In this chapter, using proper concepts and formalization, we have streamlined the details and presented a complete proof of correctness for Lang's algorithm, which has appeared in [].

To this end, we formulated the algorithm in a manner that differs from previously published descriptions. Rather than sweeping out certain edges of the crease pattern and adding the rest (the perpendiculars) as a post-processing step, we handled all edges in a unified manner and augmented the crease pattern with its event polygons. This is a conceptual device to make it easier to put the computed crease patterns into correspondence with the final 3D shapes and prove the invariants of the algorithm. We also defined and gave a classification of the final 3D shapes, called here Lang surfaces, in a manner that is independent of the algorithm. Then we showed that these flat Lang surfaces with convex boundary are precisely the subset of uniaxial origami bases that are produced by the Universal Molecule algorithm.

Open questions. Several interesting problems remain open, however. One question is whether the optimization phase of Lang's TreeMaker can be modified in order to guarantee the convexity of the resulting polygons. Alternatively, can the universal molecule algorithm itself can be modified to handle non-convex polygons? We answer this in Ch. 7 by showing that indeed it can. One goal of the present work was to clarify the properties of the algorithm so that this question can be properly tackled.

Additionally, the optimization phase of TreeMaker (the step preceding the Universal Molecule algorithm step) is known to be NP-hard [30], and the existing solutions do not always converge to a subdivision of the paper into Lang polygons. It remains open whether some other suitable method for this initial subdivision might be found which can be computed in polynomial time and which never fails.

CHAPTER 6

IMPLEMENTING THE UNIVERSAL MOLECULE ALGORITHM

In this chapter we describe and compare two efficient implementations of the universal molecule algorithm. The straightforward implementation, first given in Lang's original paper [44] and subsequent treatments computes the universal molecule in $O(n^3)$ time. The two refined implementations presented here improve the running time to $O(n^2 \log n)$: a conceptually simpler one, relying on priority queues, and a second, different approach based on a data structure called a *cyclic tournament forest*. This extends kinetic tournament trees to allow for cycle splitting operations. We compare the three implementations theoretically and in practice.

This work has appeared in [14]. We also produced a video [13] and educational website http://linkage.cs.umass.edu/origamiLang using our implementation.

6.1 Introduction

In the Ch. 5, we described the universal molecule algorithm and proved that the universal molecule crease patterns it produces are in one-to-one correspondence with the zero-curvature Lang surfaces with convex boundary. We left open, however, the running time of the algorithm.

The UM-skeleton. In this chapter we present and compare, theoretically and in practice, three implementations of the universal molecule algorithm. Instead of computing the universal molecule by tracing both the corners and the markers of the parallel sweep (as we did in Ch. 5), it is convenient to first ignore the trace of the

markers, and then add them in a post-processing step. We call the resulting structure the ***universal molecule (UM-)skeleton***. Figure 6.1 illustrates a UM-skeleton. Note that in [14] we referred to this structure as a *tree constrained skeleton*, but since it is only used in the context of the universal molecule, we find it more evocative to use the term UM-skeleton.

Contribution. We first analyze the combinatorial complexity, or number of vertices, edges, and faces, in the UM-skeleton and the universal molecule. We show that for a Lang polygon (T, P_T) where T has n nodes, the combinatorial complexity of the universal molecule is $O(n^2)$ and there exist examples with $\Omega(n^2)$ faces. This implies a trivial lower bound on the algorithm of $\Omega(n^2)$ time and space.

We then describe two improvements over the naive $O(n^3)$ time implementation. We compare, theoretically and experimentally, two implementations running in $O(n^2 \log n)$ time, without increasing the space complexity. Both make use of a data structure for storing shrinking trees which supports constant time queries and linear time split operations. The first implementation also uses a data structure based on kinetic tournament trees [] which is reminiscent of a KDS; it allows us to find the next splitting event in constant rather than quadratic time and requires sub-quadratic time to maintain at each event. While tournament trees have been used in the KDS literature, none of the applications we are aware of requires the new cyclical splitting operation introduced here. The second implementation uses a global priority queue to store all events and continually processes the top event in the queue.

Parallel sweep. Recall that the universal molecule is given by tracing the vertices of a parallel sweep polygon while a simultaneous sweep occurs in the tree (Ch. 5). The main computational difficulty arises from what might be called the "non-locality" of a splitting event. Unlike the contraction events which only happen between neighboring vertices, any given pair of vertices can, in principle, be part of a splitting event.

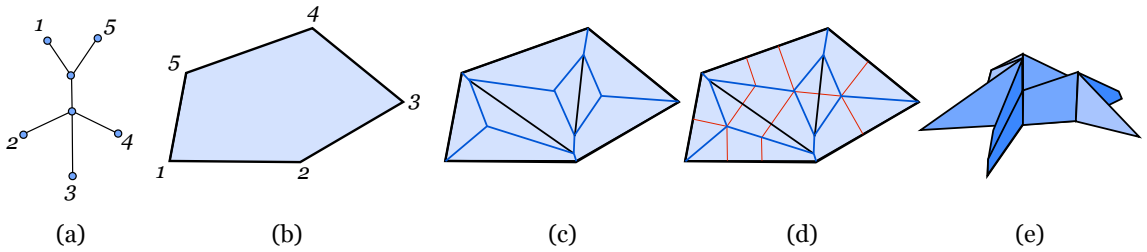


Figure 6.1: (a, b) A Lang polygon. (c) Its UM-skeleton. (d) Its universal molecule. (e) The corresponding Lang surface. The edges of the UM-skeleton form the ridge edges.

Given a pair of vertices, the time at which the Euclidean distance becomes equal to the tree distance depends a lot on the overall global geometry of the polygon, which seems to make it difficult to compute the time of the next candidate splitting event without computing over all pairs.

A further problem is that when the sweep is actually split at a splitting event, a large number of the potential splitting events that were previously computed may be invalidated. This is fairly easy to see. Suppose we have a large polygon with n nodes and a splitting event occurs between the 1st and $(n/2)$th nodes. Then all candidate events where one vertex is between the first and $(n/2)$th nodes and the other is between the $(n/2)$th and last vertex are no longer valid, thus leading to a quadratic number of invalid nodes.

Kinetic data structures. The dependence on precise metric information in a kinetic (moving) setting imposes on the UM-skeleton-algorithm the need to use substantially more (albeit still polynomial) resources over the simpler straight skeleton parallel sweep. Kinetic data structures (KDSs) were introduced in [7] for maintaining discrete properties (e.g. the convex hull) over continuously moving points. Closest to our problem of detecting and efficiently processing splitting events is the *kinetic closest pair problem* where the goal is to maintain the closest pair for a set of moving points. A KDS for maintaining the closest pair for points moving along semi-algebraic

curves is given in [7]. A *fully dynamic* KDS appears in [2] and allows for insertions and deletions of points during the simulation.

Of the KDSs we found in the literature, none appear to be readily applicable to our problem, for two reasons. First, at a splitting event, the polygon is cyclically split into two, and the sweep continues independently in each. This cyclical splitting operation is special to the UM-skeleton-algorithm and to our knowledge has not appeared in any existing KDS. Second, and more importantly, most related KDSs use special properties of the Euclidean plane and distance metrics. In the UM-skeleton case, events occur when a relationship holds between *two different metrics*: the Euclidean distance and a metric tree distance. This relationship is non-linear and does not induce a metric (it does not satisfy the triangle inequality). This makes it difficult to adapt existing KDSs to the problem. *Neither the origami, straight-skeleton, nor KDS literature contain any solution which is readily applicable to improving the running time of constructing the UM-skeleton.*

Main result. The main result of this chapter is to prove the following:

Theorem 6.1.1 (Main result)**.** *The universal molecule for a Lang polygon* (T, P_T) *with* n *nodes can be computed in* $O(n^2 \log n)$ *time and* $O(n^2)$ *space.*

Notation. As in Ch. 5, we use **bold face** to denote a vertex $\mathbf{v}$ of a polygon P_T and *italics* to denote is corresponding node v in T. We use $p_{\mathbf{v}}$ to denote its 2D coordinates and $\theta_{\mathbf{v}}$ to denote the interior angle of $\mathbf{v}$ in P_T. We denote the Euclidean distance between two vertices of P_T by $d(\mathbf{u}, \mathbf{v})$ and the tree distance by $d_T(u, v)$.

6.2 Concepts

In this section we recall a few of the concepts needed from Ch. 5 and define the UM-skeleton.

The UM-skeleton. Recall from Ch. 5 that the universal molecule of a Lang polygon (T, P_T) is defined by the trace of the vertices during a parallel sweep of P_T (and corresponding shrinking of T) together with the splitting edges introduced at a splitting event. The parallel sweep experiences two types of events, contraction events, in which an edge of the sweep contracts to a single point, and splitting events, in which a diagonal of the sweep polygon, called a ***splitting edge***, splits the sweep into two pieces and the sweep is continued independently in each.

As we have seen, the universal molecule is the subdivision of P_T induced by the traces of the vertices of the parallel sweep along with any splitting edges introduced at splitting events. Each vertex of the sweep, then, traces out an edge in the subdivision. Each edge of the sweep traces out a face. Recall, however, that the internal nodes of the tree T are *straight* in P_T, meaning that they have interior angle π. In the terminology introduced in Ch. 5, we call these ***markers***, since they are, in a sense, points "marked" on sides of the geometric polygon P_T, and are not in the strict sense vertices of P_T.

Because each edge moves in a parallel fashion throughout the sweep, it follows that the edge traced out by a marker on the sweep is always perpendicular to the sweep. Hence, we call these edges ***perpendiculars***. The remaining edges of the universal molecule are divided into two types: ***ridge edges***, which are the traces of the ***corners*** of the sweep, and the splitting edges introduced at a splitting event. We now have:

Definition 6.2.1. *The **UM-skeleton** of a Lang polygon (T, P_T) is the sub-graph of the universal molecule composed of only the ridge and splitting edges.*

The UM-skeleton sweep. We further define a new sweep based on the parallel sweep which defines the universal molecule. In this new sweep we ignore the marker vertices, and instead track contraction events only for entire sides of the polygon P_T instead of each edge of the sweep polygon. (Recall that in our terminology a *side*

of the polygon is the straight line segment connecting two corners which is further subdivided by the markers into *edges*). We call this the ***UM-skeleton*** sweep.

Because splitting events only occur between corners, the universal molecule parallel sweep and the UM-skeleton sweep differ only in the contraction events. It is easy to see that a contraction event in the UM-skeleton sweep is a contraction event in the parallel sweep–if an entire side of the polygon contracts, then necessarily all of the edges comprising that side contract. Thus, the events in the UM-skeleton sweep are a strict subset of the events of the parallel sweep defining a universal molecule.

We note that the UM-skeleton sweep has the same relationship to the UM-skeleton as the parallel sweep does to the universal molecule. Namely, the trace of each vertex of the UM-skeleton sweep together with the splitting edges form the edges of the UM-skeleton and the trace of each edge in the sweep form the faces of the UM-skeleton. This fact is useful in the next section in establishing the combinatorial complexity of both the UM-skeleton and the universal molecule.

Purpose of the UM-skeleton. We are primarily interested in the UM-skeleton for two reasons. The first is that as we have seen the number of contraction events that occur in the UM-skeleton sweep is less than or equal to the number of contraction events that occur in the parallel sweep of a Lang polygon. In fact, we will see in the next section that for a Lang polygon (T, P_T) where T has n nodes, the number of events that occur in the UM-skeleton sweep is $O(n)$, whereas the number that occur in the parallel sweep of the universal molecule is $O(n^2)$, and there exist examples requiring $\Omega(n^2)$ contractions. The second reason we are interested in the UM-skeleton is that it simplifies the complexity analysis given in the next section.

6.3 Combinatorial Complexity

We now initiate a study of the combinatorial complexity of the UM-skeleton of a Lang polygon (T, P_T) and then use this to analyze the complexity of the univer-

sal molecule. This analysis establishes lower bounds for the running time of any algorithm computing the universal molecule.

Complexity of the UM-skeleton. Recall from the previous section that the faces of the UM-skeleton are given by the traces of the edges of the sweep. Let l denote the number of leaf nodes in T. Then P_T has l corners and l sides. Initially, then, the sweep is tracing out l faces. Only two types of changes occur in the sweep. At a contraction event, one side of the sweep contracts to a single point, and thus that face is "completed", in the sense that the sweep leaves the face (and does not return). The second is a splitting event which splits the sweeping polygon into two along a diagonal. This "splitting edge" diagonal is really two new sides to the sweep–one in each of the split polygons. Thus, a splitting event introduces two new sides to the sweep, which trace out two new faces. We now prove:

Lemma 6.3.1. *Let (T, P_T) be a Lang polygon and let l denote the number of leaves in T. The UM-skeleton sweep encounters $O(l)$ events.*

Proof. Specifically, we prove that The UM-skeleton (T, P_T) has $l + 5m + k$ edges and $l + 2m$ faces where $m = O(l)$ is the number of splitting events and $k = O(l)$ is the number of side contraction events.

Each face is "completed" by a polygon side or splitting segment contracting in the parallel sweep and each side eventually contracts. There are l sides in P_T and each splitting event adds two new splitting segments to trace (one in each of the left and right split polygons). Thus there are $l + 2m$ faces. To count the edges in the UM-skeleton observe that initially the sweep traces l corner vertices. At each of the k contraction events the trace of two (or more) corners meets at a point which becomes a new corner in the sweeping polygon. The trace of this corner is a new ridge edge. At a splitting event one splitting segment is added to the UM-skeleton between the splitting pair. Both corners of the splitting event have copies in each

of the split polygons. Thus a splitting event introduces one split edge and four new ridge edges. Therefore there are $l+5m+k$ edges in the UM-skeleton. That $m = O(l)$ is simply the fact that a polygon with l sides can be recursively divided by dividing along non-crossing diagonals at most $O(l)$ times. □

As a direct corollary of Lemma 6.3.1, we have:

Corollary 6.3.2. *Let (T, P_T) be a Lang polygon and let l denote the number of leaves in T. The UM-skeleton has $O(l)$ vertices, edges, and faces.*

This bound is obviously tight, since we must at least have l faces, one for each of the initial sides of P_T.

Complexity of the universal molecule. We now look at "adding" the perpendiculars back to the UM-skeleton to obtain the universal molecule. Each perpendicular is the trace of one of the markers on a side of the sweep. By definition, a side of P_T corresponds to a path in T between leaf nodes. Thus a marker corresponding to a given internal node in T appears at most once on any given side of the sweep. It follows, then, that each internal node of T gives rise to at most one extra marker edge on the interior of each face of the UM-skeleton. Thus we have:

Corollary 6.3.3. *Let (T, P_T) be a Lang polygon and let n denote the number of nodes in T. The universal molecule has $O(n^2)$ vertices, edges, and faces.*

Constructing a worst-case. We now show that this bound is tight by constructing a family of examples using the Lang surfaces defined in Ch. 5. We note that the family we construct is highly degenerate, in that multiple splitting events occur simultaneously in the resulting sweeps and the tree has degree 2 nodes; however, even if we disallow such degeneracies it is possible (with a little care) to obtain examples that are essentially the same as this one. We present this one because of its simplicity and leave the construction of other examples as an exercise to the reader.

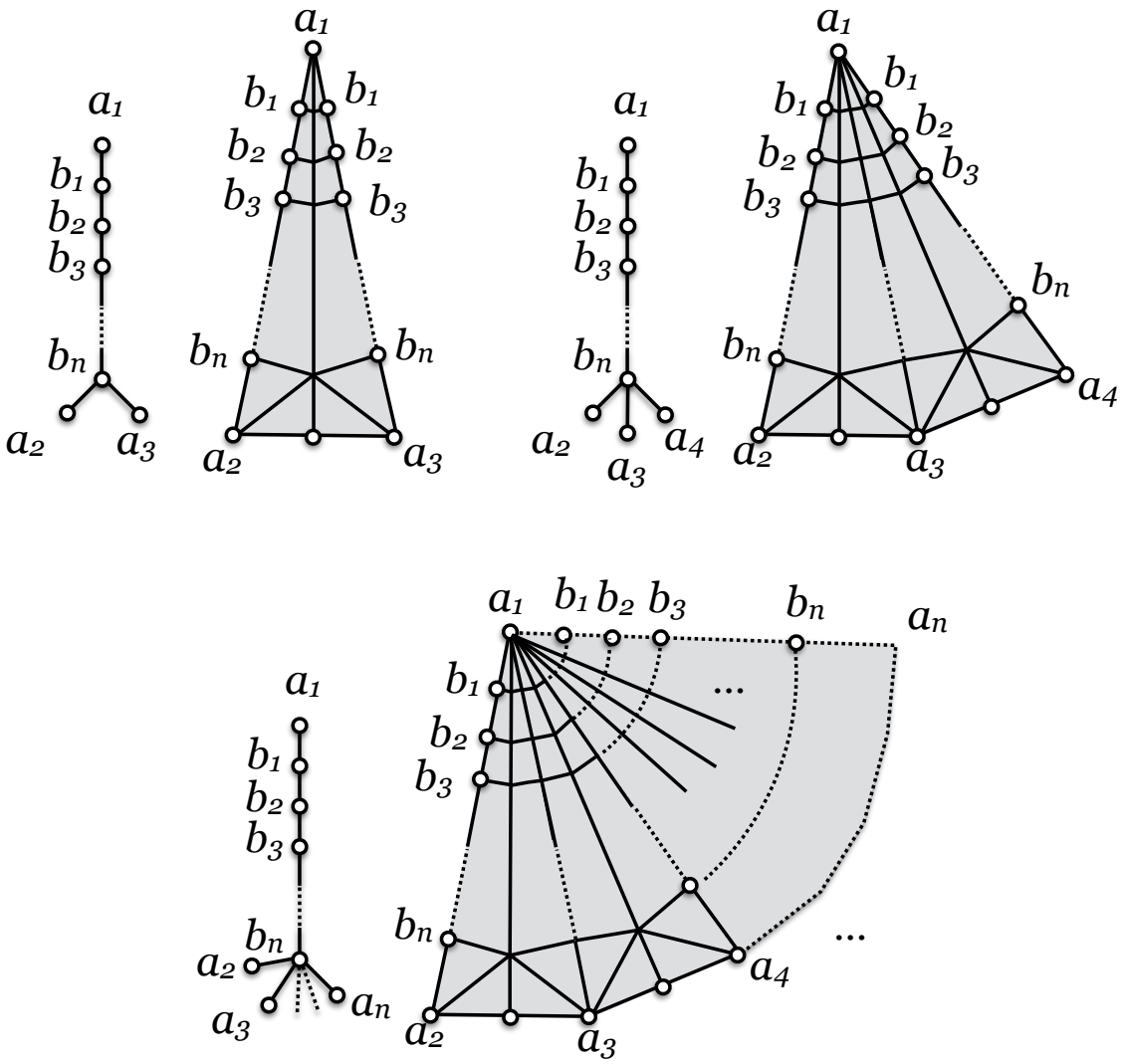


Figure 6.2: A UM-skeleton (a) and partial UM-skeleton (b) The blue are ridge edges and black are splitting edges. Gray faces in (b) represent the parts of the polygon not yet encountered by the sweep.

The basic construction is illustrated in Fig. 6.2. We start with a tree T that has three leaf nodes a_1, a_2, a_3 and n internal nodes $b_1, \ldots, b_n$. The leaf arc incident to a_1 is also incident to b_1 and the two leaf arcs incident to a_2 and a_3 are incident to b_n. All arcs have the same length. Since T has only three leaf arcs, then it has a unique polygon P_T with which it forms a Lang polygon. P_T is a base case of the universal molecule algorithm (since it is a triangle) and the universal molecule for P_T has $2n+4$ faces. We now use the gluing operations on Lang surfaces from Ch. 5 to form a Lang polygon with 4 leaves and n internal nodes. Simply glue a copy of T to itself by gluing the first copy along the path from a_1 to a_3 in the tree to the second copy along path from a_1 to a_2. From this we obtain a tree with n internal nodes and 4 leaf nodes (see Fig. 6.2 top left). Similarly, to form the polygon, we take P_T and copy it. Then rotate the copy around a_1 to align the $\mathbf{a}_1\mathbf{a}_2$ side of the copy with the $\mathbf{a}_1\mathbf{a}_3$ side of the origina (again see Fig. 6.2 top left). We then relabel as in Fig. 6.2 top left. Note that because the distance from $\mathbf{a}_1$ to $\mathbf{a}_3$ is preserved, and was originally a side of P_T, the first operation performed by the universal molecule is to split P_T into two copies of the original triangle. The resulting crease pattern is then the same as the original, except copied once. Thus we have added exactly one node to the tree, a_4, but have added $2n + 4$ faces to the universal molecule. If we continue applying this operation another $n - 2$ times, the result is a tree with n leaf nodes and n internal nodes, a polygon with $4n + 2$ edges, and a universal molecule with $2n^2 + 4n$ faces.

As a corollary of the discussion above we have:

Corollary 6.3.4. *Any algorithm computing the universal molecule of a Lang polygon (T, P_T) requires $\Omega(n^2)$ time and space (where n denotes the number of nodes in T).*

6.4 Algorithm and data structure preliminaries

Recall that in Ch. 5 we simulated the sweep via a recursive algorithm. Finding the next contraction event amounts to checking, for each edge in the sweep, the time

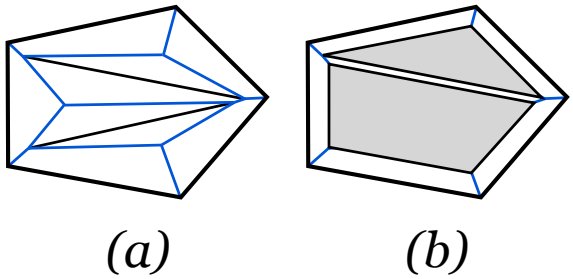


Figure 6.3: A UM-skeleton (a) and partial UM-skeleton (b) The blue are ridge edges and black are splitting edges. Gray faces in (b) represent the parts of the polygon not yet encountered by the sweep.

at which it contracts and taking the minimum. Finding the next splitting event requires checking all pairs of non-consecutive vertices $(\mathbf{u}, \mathbf{v})$ for the time at which it first becomes true that $d(\mathbf{u}, \mathbf{v}) = d_T(u, v)$.

Partial UM-skeleton and contours. If we stop the UM-skeleton sweep early, we have a planar graph made up of four types of edges: the original polygon edges, splitting edges already encountered by the sweep, the traces of the sweep vertices traced so far, and current sides of the sweeping polygons. We call this planar graph a *partial UM-skeleton* and note that it is equivalent to the final UM-skeleton with some number of polygonal "holes" cut out of it where each hole has empty interior and corresponds to one of the current contours. See Fig. 6.3(b). We call the faces of a partial UM-skeleton that represent the current sweeping polygons ***contours***.

Storing the planar graph. As in Ch. 5, we simulate the sweep recursively. This requires two data structures: one for storing a partial UM-skeleton (a planar graph), and another for storing the shrinking trees obtained after several splits. On the planar graph we require an operation to advance the sweep in a contour to time t and compute the trace of its vertices, and an operation for splitting a face by adding a splitting edge. We use the *doubly-connected edge list* (DCEL) [35] which supports the sweep advance in linear time and the contour splitting in constant time. We store additional information at the vertices (and refer to this as a "label"); for each

polygon vertex $\mathbf{v}$, the label contains its coordinates p_v; and we keep a *tree label* v for the corresponding leaf node in the shrinking tree. The content of v depends on the tree implementation (we will see several below). If v is incident to a sweep polygon, we also label it with its interior angle $\theta_{\mathbf{v}}$ and its motion vector $b_{\mathbf{v}}$.

Given the labels at a vertex, an appropriate representation of the tree, and a pair of vertices $(\mathbf{u}, \mathbf{v})$ the time at which the Lang property holds with equality for the pair is given by solving for t in:

$$||(p_v + tb_v) - (p_u + tb_u)|| = d_T(u, v) - t(\cot(\theta_{\mathbf{u}}/2) + \cot(\theta_{\mathbf{v}}/2)) \tag{6.4.1}$$

Storing a shrinking tree. For the tree we need a data structure which allows us to query, in constant time, the distance in the tree for pairs of leaves, and supports the following operations. SPLIT(i, j), which splits the tree between leaves i and j, and UPDATE(t), which updates (shrinks) the tree to time t with the pre-condition that t is less than or equal to the next contraction time. At the end of Sec. 6.4.1 we briefly discuss a straightforward implementation which supports each of these operations in $O(n^2)$-time. We improve this in Sec. 6.5 by providing a data structure which supports each operation in $O(n)$-time.

6.4.1 Algorithm Overview

We now briefly review the UM-skeleton-algorithm. What follows is essentially a representation of the algorithm from Ch. 5, but with slight modifications to compute the UM-skeleton. The *input* is a Lang polygon (T, P_T) and the *output* is the UM-skeleton G for P_T. The polygon P_T is given as a (ccw) list of 2D points labeled with the corresponding nodes in T. The tree T is given as a list of nodes, a list of edges (each labeled with its weight), and an ordered adjacency list at each node. An example run of the algorithm is illustrated in Fig. 6.4. The algorithm works by

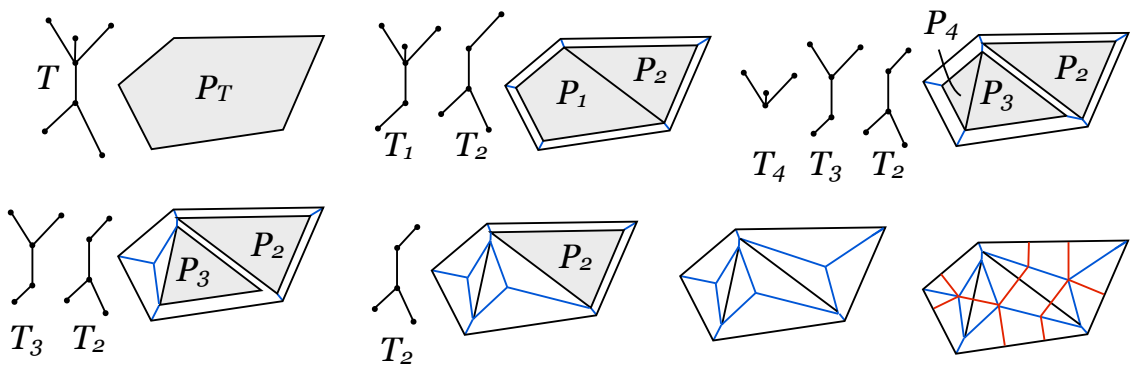


Figure 6.4: An example run of the algorithm. The shaded faces denote the contours in the partial UM-skeleton G as it is built. The first event is a splitting event which splits T and P_T into T_1, T_2, and P_1, P_2. The second event is a splitting event of T_1 and P_1 into T_3, T_4 and P_3, P_4. Next is a complete contraction (base case) in T_4, P_4. Then a complete contraction of P_3. Then a complete contraction of P_2. The bottom right figure is the universal molecule given by the UM-skeleton with perpendiculars (in red) added by the post-processing step of Lang's universal molecule algorithm.

simulating the sweep in P_T and recursively building a partial UM-skeleton G. Each contour in G represents an as yet unprocessed event. The algorithm recursively takes as input a contour in G and its corresponding shrunken tree and recursively simulates the sweep process on the contour's interior. To initialize the algorithm we initialize G with P_T as its only face and call UM-Skeleton(T, P_T, G). The algorithm follows two basic steps: first, it detects the next event; and second, it acts on the next event by advancing the sweep in the input contour and if the event is a splitting event, splitting the contour and tree. It then recurses on the new contour(s).

Main operations. FindNextEvent returns the time of the next event, and, if it is a splitting event, the splitting pair (u, v). AdvanceSweep advances the sweep in the tree and contour to the next event and computes the trace of the corners between the current and next contour. This returns a new shrunken tree and contour T' and P'_T representing the advanced sweep. Finally, if at a splitting event SplitSweep splits the new tree T' and contour P'_T by the splitting pair. The algorithm is then recursively applied to the new contour(s). As in Ch. 5 base case occurs when the

Algorithm 2 UM-SKELETON(T, P_T, G)

```
if BASECASE(P_T) then
 return HANDLEBASECASE(P_T, G)
end if
E, (u, v) ⟵ FINDNEXTEVENT(T, P_T)
(T', P'_T) ⟵ ADVANCESWEEP(T, P_T, G, H)
if event E is a splitting event then
 P_L, T_L, P_R, T_R ←SPLITSWEEP(T',P'_T,(u,v),G)
 UM-SKELETON(T_L, P_L, G)
 UM-SKELETON(T_R, P_R, G)
else
 UM-SKELETON(T', P'_T, G)
end if
```

contour contracts to a degenerate polygon: a single vertex or a two-sided polygon. We now prove the following useful lemma, which we use to analyze the complexity:

Lemma 6.4.2. *If the initial metric tree has n nodes then, at any point in the algorithm, the partial UM-skeleton has $O(n)$ vertices incident to any contour.*

Proof. We prove this by induction on the number of events encountered at any point in the algorithm. Let m be the number of splitting events encountered at some point in the algorithm and k be the number of vertices in G which are incident to a contour. We show that $k \leq n+2m$ where n is the number of leaf nodes in the input tree. Initially no events have been encountered (so $m = 0$) and $k = n$ which establishes the base case for induction. Now assume, for inductive hypothesis, that we have encountered i events, m of which are splitting events and $k \leq n+2m$. We will show that after encountering event $i+1$, the number of vertices k' incident to a contour in G is $\leq n+2m'$ where m' is the number of splitting events encountered at event $i+1$. Event $i+1$ is either a contraction event or a splitting event. In the first case $m' = m$ and since we have a contraction of at least one side to a vertex $k' < k$. Thus $k' < k \leq n+2m = n+2m'$ by inductive hypothesis. In the second case $m' = m+1$ and two new vertices are added.

Thus $k' = k+2$. So $k' = k+2 \leq n+2m+2 = n+2(m+1) = n+2m'$ (by inductive hypothesis). Since $m = O(n)$ (as discussed in Sec. 6.3) this proves the lemma. □

Operation details. FINDNEXTEVENT(T, P_T) find the next candidate contraction and splitting events. The next candidate contraction event can be found naively in $O(n)$ time by checking the time at which each edge contracts along its current trajectory. As before the next candidate splitting event is the minimum time t solving Eq. 6.4.1 over *all* $O(n^2)$ *pairs* of non-consecutive vertices in P_T. From the DCEL we obtain the list of vertices in $O(n)$-time. Each time Eq. 6.4.1 is solved, it requires a single call to QUERY in the tree. ADVANCESWEEP(T, P_T, G, H) moves the sides of P_T inwards and contracts any zero-length edges which result. The DCEL handles this in $O(n)$-time. The tree is then shrunk by a call to UPDATE. The SPLITSWEEP$(T', P'_T, (u, v), G)$ operation splits the tree T' and contour P'_T between (u, v) and (t_u, t_v) (resp.). Splitting P'_T is handled in $O(1)$-time in the DCEL, and splitting the tree is handled by calling SPLIT on T'. To complete the complexity analysis, we need to provide an implementation of the tree data structure. The straightforward implementation and analysis follows.

Straightforward tree structure. We now briefly discuss a straightforward approach to storing the tree which leads to $O(n^2)$-time operations and analyze the complexity of the algorithm using this data structure. In Sec. 6.5 we replace this with a data structure supporting $O(n)$-time operations. Store the tree T for each contour P_T as a 2D array D, where each entry D_{ij} is the distance between leaves i and j in T. QUERY(i, j) returns the value D_{ij}. The UPDATE and SPLIT operations are basic matrix operations and each require $O(n^2)$-time.

UPDATE(h) ranges over all pairs of non-consecutive leaves (i, j) and updates D_{ij} to $D_{ij} - h(s_i + s_j)$. Note that this allows arcs to collapse to zero-length, but does not remove them.

SPLIT(i, j) takes two non-consecutive leaf nodes i and j and returns arrays D_L and D_R representing the left and right trees obtained by splitting between i and j. D_L (resp. D_R) is filled with the entries of D corresponding to all pairs of leaf nodes (k, l) such that (cyclically) $j <= k, l <= i$ (resp. i and j).

Initialization and complexity. The DCEL G is initialized with a single face P_T. The initial entries of D are found by breadth-first search (BFS) from each leaf of T. The analysis is given in Lem. 6.4.3.

Lemma 6.4.3. *Given the straightforward tree representation* UM-SKELETON *takes $O(n^3)$-time and $O(n^2)$-space.*

Proof. The time complexity follows directly from the operation and data structure details above. Each edge or vertex of the DCEL G is either a vertex or (possibly part of) an edge in the final output or is incident to a contour. By Lemmas 6.3.1 and 6.4.2, G has $O(n)$ vertices, edges, and faces each requiring constant storage. We also need to store the distance matrices for each contour. By Lemma 6.4.2 we have $O(n)$ nodes incident to any sweep face and thus $O(n^2)$ entries in total over all matrices. □

Bottlenecks. There are two bottlenecks in the straightforward implementation: (1) FINDNEXTEVENT's checking of all non-consecutive pairs of vertices and (2) the $O(n^2)$-time tree operations, UPDATE and SPLIT, which maintain the tree data structure. The rest of this chapter is concerned with removing these bottlenecks to improve the running time of the algorithm to $O(n^2 \log n)$-time.

6.5 Representing Shrinking Trees Implicitly

The first improvement comes by replacing each shrinking tree's distance matrix with an implicit representation and a single global distance matrix. This improves the main operations on the tree data structure from quadratic to linear time while

keeping the same constant-time query operation and quadratic-space requirement. We use a single immutable distance matrix which stores the *initial* distances between *all leaf of nodes* in the tree and a set of labels corresponding to leaves of the subtree. We maintain a single array D where each D_{ij} is the distance in the *initial* tree T between nodes i and j. We assume each leaf i has a known speed s_i.

Representing sub-trees implicitly. To represent a sub-tree we use a list of leaf-markers $L = (l_1, \ldots, l_k)$. A ***leaf-marker*** $l = (N, d)$ is given by a leaf node N of the initial tree T and the distance d the leaf has moved from T'. The shrunken tree is given by the union of the paths in T between the leaf nodes of all pairs of leaf-markers in L, which the metric on each leaf arc decreased by the appropriate distance d.

An example. As an example, suppose we have a tree T with four leaf nodes all connected to a single internal node by length 1 arcs. Denote the leaf nodes by (a_1, a_2, a_3, a_4) and the internal node by b. Now suppose we want to represent the sub-tree of T given by the union of paths between all pairs of the leaf nodes (a_1, a_2, a_3) where the length of the arc incident to a_1 has shrunk to 0.8, the length of the arc incident to a_2 is 0.4, and the length of the arc incident to a_3 is 0.1. Then our implicit representation $L = ((a_1, 0.2), (a_2, 0.6), (a_3, 0.9))$. We could, in principle, recover the tree from this representation, but we will see shortly that we do not need to in order to support the desired operations.

The operations. The distance between two leaf nodes i and j in T' is given by the distance between the leaf nodes in T minus the distances each have moved from T to T'. For two leaf nodes i and j with leaf-markers (N_i, d_i) and (N_j, d_j), QUERY(i, j) returns $D_{N_i,N_j} - (d_i + d_j)$. UPDATE$(t)$ simply adds ts_i (where s_i denotes the speed assigned to leaf i) to d_i. SPLIT(i, j) splits the list L into L_{left} and L_{right}. Each leaf-marker l_k is placed in L_{left} (resp. L_{right}) if (cyclically) $j \leq k \leq i$ (resp. $i \leq k \leq j$). This takes $O(n)$-time.

Initialization. The distance matrix D is initialized by BFSs from each node of T in $O(n^2)$-time. The initial set of leaf-markers L is given by creating a marker $(i, 0)$ for each leaf i in T which takes $O(n)$-time.

Complexity. This representation removes one bottleneck (see the end of Sec. 6.4.1) related to the tree representation: AdvanceSweep and SplitSweep are improved from $O(n^2)$ to $O(n)$ time. The space requirement remains $O(n^2)$-space. We now turn to the remaining bottleneck: the quadratic-time search for the next splitting event.

6.6 Cyclic Tournament Bushes

The remaining bottleneck is the quadratic-time search for the next splitting event. For the first implementation we adapt a tool commonly used in the construction of KDSs called a *kinetic tournament tree* first used in [7] for maintaining the minimum of a list of changing values over time. In our setting the data set is a cyclical list of items which needs to be split into two at each splitting event. This cyclicity is a specialized property which comes from the fact that our "item lists" are the ccw vertices incident to a face. We exploit an *ordering property* of the tournament tree to add an $O(\log n)$-time cyclical splitting operation. In the next section we show how to use these trees to find splitting events in $O(n \log n)$-time.

Note that this splitting operation differs from the usual splitting operation on trees. Typically split operations on binary search trees split the trees by value. In other words, a tree T is split into two trees T_1 and T_2 such that all of the values of T_1 are less than any value in T_2. Our split operation, however, operates on the original indices of the values in the list used to build the tree. It specifically does not split by value–both trees T_1 and T_2 may contain values which are less than some values in the other.

Tournament trees. A tournament tree is a heap-like full binary tree for finding the minimum value of a set of items $1, \ldots, n$ with values $v_1, \ldots, v_n$. Conceptually, a tournament tree is a tournament between "competitors" representing the items. Each level of the tree is a round, the leaves are competitors, and the internal nodes are bouts. The smaller valued competitor wins each bout and progresses upwards through the tree to play at the next level. Each internal node represents the winner of all leaves in its sub-tree. See Fig. 6.5(a). The tree has $2n - 1$ nodes and is initialized from the leaves up in $O(n)$-time. We observe that a tournament tree also maintains an *ordering property*: the left-to-right order of the leaves corresponds to the order on the items.

The kinetic version of a tournament tree also has an UPDATE operation, which updates the tree when the value at a leaf node changes. When any node's value changes its parent node may become invalid, and thus the parent's value must change. The operation updates the leaf node and then recursively propagates the change upwards through all ancestors to the root. In [2] the kinetic tournament tree was extended to the dynamic setting with *insert* and *delete* operations. They relax the requirement that the tree remain full, and add additional INSERTLEAF and DELETELEAF operations to the tree. Supporting the INSERTLEAF operation requires rebalancing the tree. In our setting only DELETELEAF is required, and we are able to avoid rebalancing the tree[1].

Cyclic tournament bushes. In addition to the DELETELEAF operation, we define an $O(\log n)$-time SPLIT(i, j) operation which splits the tree into a tree for items (cyclically) j to i, and a tree for items i to j. We call the resulting tree data structure along with the UPDATE, DELETELEAF, and SPLIT operations a ***cyclic tournament bush*** (CTB). Conceptually, a CTB is a tournament where some competitors do not

[1]In principal our method can be extended to use rebalancing, but doing so unnecessarily complicates the exposition and does not effect the overall running-time of the UM-skeleton algorithm.

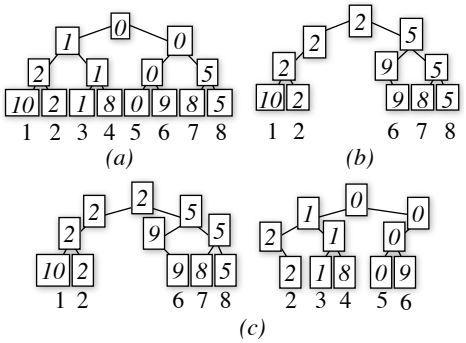


Figure 6.5: (a) A tournament tree. (b) A CTB for the same competition where 3, 4, and 5 do not show up. (c) The result of the SPLIT$(2, 6)$ operation on the tree in (a).

show up; we call it a *bush* instead of *tree* to indicate this feature. If only one competitor shows up to a particular bout, it is declared the winner. See Fig. 6.5(b). The CTB maintains the heap and ordering properties. Additionally, we store the index of the *leftmost* and *rightmost* descendant leaves at each internal node so that the leaf for a particular item can be found by an $O(\log n)$-time search.

Splitting a CTB. The SPLIT operation on a CTB B for an ordered list of items $1, \ldots, n$ takes as input two items i and j and produces two CTBs B_1 and B_2 such that the items for B_1 are (in order) $1, \ldots, i, j, \ldots, n$, the items for B_2 are $i, \ldots, j$, and the depth of B_1 and B_2 is equal to the depth of B. In order to make the operation fast, we make it destructive. Obviously, if we kept the initial input B intact and returned the two entirely new CTBs B_1 and B_2, the operation would have a trivial $\Omega(n)$ lower bound. To reduce the time required by the operation to $O(\log n)$, we destroy the input B by reassigning parent pointers for nodes along the paths from the root of B to i and j and make copies only of the nodes along this path.

Split operation details. The operation performs the following steps:

1. Create empty trees B_1 and B_2 and copy the path from i to j (through the root) into each.

2. For each node N on the path from i to $root(B)$, either the left child N_L or the right child N_R is on the path. If N_L is, reset the parent of N_R to the copy of N in B_2, otherwise reset the parent of N_L to the copy of N in B_1. Similarly, for each node N on the path from j to $root(B)$ if N_L is on the path, reset N_R's parent to B_1, otherwise reset N_L's parent to B_2.

3. Check each node in B_1 and B_2 along the copies of the paths from i and j to $root(B)$ to check whether its winner is still correct. If not, update its winner.

At the end of this operation, B_1 and B_2 are CTBs for the list of items $1, \ldots, i, j, \ldots, n$ and $i, \ldots, j$ resp. See Fig. 6.5(c). An illustration of each step of this operation is provided in Fig. 6.6.

Correctness and analysis. Correctness follows from the ordering property on the tree, and the observation that the only nodes in B_1 and B_2 which have a different sub-tree than their copies in B are those nodes along the copied path from i to j. The time complexity of the operation is $O(d)$-time where d is the depth of the tree. However, we note that if we begin with a full CTB, since each operation preserves its depth, the splitting operation takes $O(\log n)$-time where n is the size of the initial list.

Summary. In this section we presented a tree data structure for maintaining the minimum value of a set of items with $O(\log n)$-time UPDATE, DELETELEAF, and (cyclical) SPLIT operations. In the next section we show how to use this data structure to improve the quadratic-time search for splitting events in the UM-algorithm to $O(1)$ time with $O(n \log n)$ time maintenance.

6.7 The Cyclic Tournament Forest Implementation

We now show how to improve the quadratic-time search for splitting events to $O(1)$ time using a data structure built from the CTBs of the last section. We use

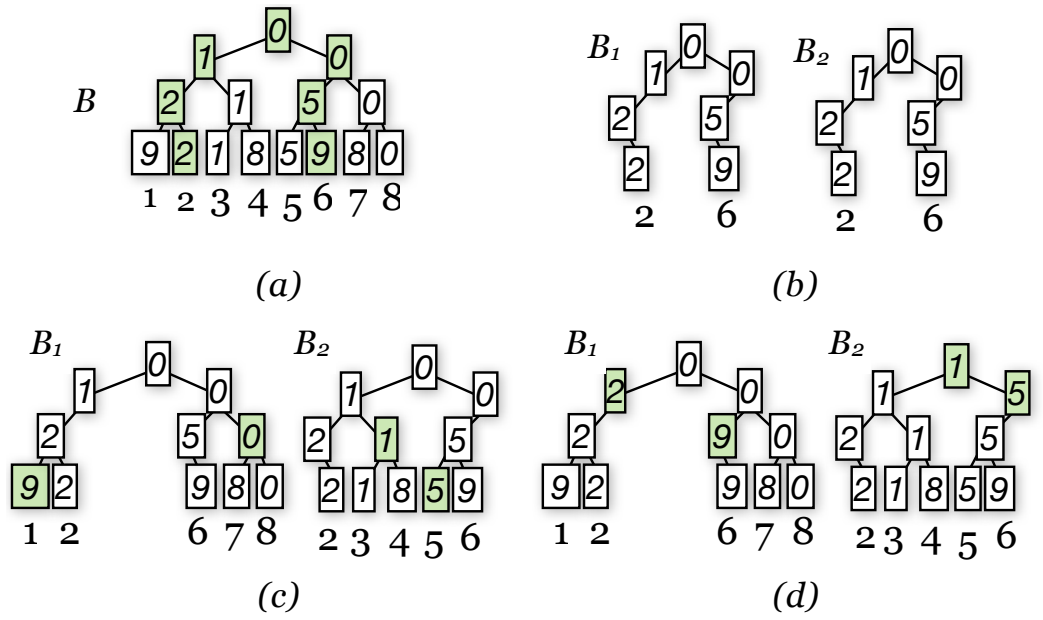


Figure 6.6: The SPLIT$(2, 6)$ operation on the CTB B in (a). (b) The splitting path is copied into B_1 and B_2. (c) The sub-trees are moved to B_1 and B_2 by changing the parents of the green shaded nodes. (d) The winner of each bout on the splitting path is updated. The green shaded nodes are the nodes which have different values than their copies in B.

the following abstraction of the search. Let $I = (1, ..., n)$ be a list of items and v_{ij} be values defined for each (unordered) pair of items (i, j).

In this section we give a data structure which supports the following operations.

- FINDMIN, which returns the smallest value v_{ij}.
- UPDATEVAL(i, j, val) which sets v_{ij} to val.
- DELETEITEM(i) which deletes item i.
- SPLITLIST(i, j) which splits the data structure into structures for maintaining the minimum values v_* over $I_1 = (1, ..., i, j, ..., n)$ and $I_2 = (i, ..., j)$.

We call our structure a ***cyclic tournament forest (CTF)***. We call our structure a *forest* because it is constructed by creating a cyclic tournament bush for each item in i.

Cyclic tournament forests. For each item i in I, we store a CTB B_i on the list of values $(v_{i1}, ..., v_{in})$. In other words, the CTB for each item stores the values defined

between item i and every other item j. We then store the list $(B_1, ..., B_n)$ of the CTBs for all the items itself in a CTB F, where the "value" of each B_i is given by the value of its root.

Operations. The operations are implemented as follows.

- FindMin simply returns the value of the root node of the CTB B_i returned by Query on F.
- UpdateVal(i, j, val) performs the following:
  1. Call Update(v_{ij}, val) on B_i and B_j.
  2. If the root node of either, say B_i, changes its value as a result of the Update, call Update(B_i, val) on F (resp. Update(B_j, val)).
- DeleteItem(i) first calls DeleteLeaf(v_{ik}) on all B_k where $k \neq i$ and then calls DeleteLeaf(B_i) on F.
- The SplitList(i, j) operation builds two lists of CTBs, L_1 and L_2 by the following procedure. For each item k (in order) let B' and B'' be the CTBs obtained by splitting B_k between i and j. If $k \in I_1$, add B' to L_1. If $k \in I_2$, add B'' to L_2. To save space, we modify the split operation on B_k so that if $k \neq i, j$, only one of B' and B'' is built depending on whether $k \in I_1$ or $k \in I_2$. The operation then builds two full CTBs, F_1 on L_1 and F_2 on L_2 and returns them. The original F is deleted.

Correctness. The correctness of FindMin, UpdateVal, and DeleteLeaf is apparent. SplitList works because each tree B_k is built on the list of items $(v_{k1}, ..., v_{kn})$. The Split operation on B_k produces the CTBs B' and B'' built on the lists $(v_{k1}, ..., v_{ki}, v_{kj}, ..., v_{kn})$ and $(v_{ki}, ..., v_{kj})$, resp. Thus if $k \in I_1$, B' contains exactly the values

between k and any other item in I_1 (similarly for $k \in I_2$ and B''). The correctness follows.

Complexity analysis. FindMin checks the value of the root of F in constant time. UpdateVal makes a constant number of calls to Update, and thus takes $O(\log n)$-time. Each of the remaining two operations make $O(n)$ calls to $O(\log n)$-time CTB operations; additionally, the SplitList operation initializes two new CTBs in $O(n)$ time. Thus both operations take $O(n \log n)$-time. In terms of space, only the SplitList operation creates any nodes: a copy of the $O(\log n)$-size splitting paths are made for B_i and B_j, and the new trees F_1 and F_2 are initialized. Let m be the number of items in F_1, then $n - m + 2$ is the number of items in F_2. Since F_1 and F_2 are full CTBs, the total number of nodes in F_1 and F_2 is $2m+2(n-m+2)-2 = 2n+2$, which is only 3 more nodes than F. F itself is deleted. Thus, each Split operation adds a total $O(\log n)$ nodes.

Finding splitting events with cyclic tournament forests. We now use the cyclic tournament forest F defined above to accelerate the process of finding splitting events. We maintain a forest F for each contour P_T. The *list of items* of F is the list of vertices of P_T given in ccw order. Each value v_{ij} for non-consecutive pairs (i, j) is equal to the time t solving Eq. 6.4.1 for vertices i and j. We denote this by t_{ij} for a pair (i, j). For (i, j) consecutive we set $v_{ij} = \infty$. The next splitting event for P_T is found by FindMin on F.

Maintaining the cyclic tournament forests. When we process a splitting event (i, j) of P_T into P_1 and P_2, we also split F into F_1 and F_2 by the following procedure.

1. Call SplitItems(i, j) on F to obtain F_1 and F_2.
2. Call UpdateVal(i, k, t_{ik}) and UpdateVal(j, k, t_{jk}) on F_1 and F_2 for each k in P_1 and P_2 resp.

The correctness of the maintenance procedure follows from the observation that when we split P_T, the angle bisector for any $k \neq i, j$ in P_1 or P_2 does not change. Given two such vertices k_1, k_2 of P_1 (resp. P_2), the value $t_{k_1 k_2}$ is the same after the split as before it. However, for i and j, the bisectors do change, and thus for any k, t_{ik} and t_{jk} have changed. The call to SplitItems guarantees that P_1 and P_2 have the appropriate structures, but the observation above shows that because i and j have new angle bisectors in P_1 and P_2, the values v_{ik} and v_{jk} need updating. This is handled in step 2 by the UpdateVal procedure. Since this requires $O(n)$ calls to UpdateVal and a single call to SplitItems, maintenance of the cyclic tournament forest requires $O(n \log n)$ time and creates $O(\log n)$ nodes per split event. Given this we prove:

Theorem 6.1.1. *The UM-skeleton of a Lang polygon w.r.t. a metric tree with n nodes can be computed in $O(n^2 \log n)$-time and $O(n^2)$-space.*

Proof. *Time:* Initialization requires $O(n)$-time for the DCEL and implicit tree and $O(n^2)$-time for the CTF. By the discussion above and the operation details in Sec. 6.4.1 we have the following. FindNextEvent finds contractions in $O(n)$-time and splitting events in $O(1)$-time. The running time per event is dominated by the $O(n \log n)$-time maintenance of the CTF. By Lemma 6.3.1, there are $O(n)$ events, and thus the algorithm takes $O(n^2 \log n)$-time. *Space:* The DCEL requires $O(n)$-space (see proof of Lem. 6.4.3). For the implicit trees we store the $n \times n$ matrix D and a constant size leaf-marker at each vertex of G. We store a CTF for each contour, starting with one $O(n^2)$-space CTF. Each splitting event results in the creation of $O(\log n)$ new nodes. For $O(n)$ events, the entire running of the algorithm creates $O(n \log n)$ nodes. Total space is thus $O(n^2)$. □

6.8 The Priority Queue Implementation

We now briefly describe a priority queue implementation. We use the same implicit tree representation described in Sec. 6.5. We maintain a priority queue Q of

potential events. Each event stores a ***type***, either *splitting*, *contraction*, or *base* and the time t at which the potential event occurs. We store each sweep polygon as a (cyclic) list of vertices.

The basic idea is to query the priority queue to obtain the next event. A technicality to this approach, however, is how to deal with the fact that after a split, the priority queue may contain many potential splitting events that are no longer valid (i.e. one vertex of the potential event is on one side of the split, the other vertex is on the other side). Recall from the discussion in Sec. 6.1 that the number of invalidated potential events may be quadratic. Rebuilding the priority queue or removing these invalid events would require $O(n^2 \log n)$ queue initialization or maintenance procedures at each splitting event. Thus, to gain efficiency, we need a way of avoiding deleting invalid events.

Ignoring events. Our approach to dealing with the invalid splitting events is simply ignore them–just leave them in the queue. We will see that this does not add too much extra processing. In order to deal with these events we change the structure of the algorithm slightly. We maintain a list of *active contours* which are the contours we described in Sec. 6.4.1. The idea is that we "move" vertices in the sweep. For each vertex, we store a pointer to the contour it is currently active in and a boolean flag *active* which signals whether the vertex is still part of an active sweep polygon.

In the queue, for a contraction event we store a pointer to an edge (u, v) of a sweep polygon. For a splitting event we store pointers to the splitting pair (u, v). A base event stores a pointer to the sweep polygon P which is a base case.

Modifications to the algorithm. We modify the algorithm to use the queue instead of proceeding recursively. At each iteration until Q is empty, we remove the next event from Q for processing. If the event is a base case, we simply process it using HandleBaseCase and continue. Otherwise, we first check that u and v are

still active and that they point to the same sweep polygon. If not, then the event is invalid, and we throw out the event and continue. This occurs because at some point $\mathbf{u}$ and $\mathbf{v}$ were separated by a splitting event, and thus their *active contour* pointers point to different contours.

If we have a valid event, then we advance the sweep P by moving its vertices inwards along the angle bisectors to the time of the event. If the event is a contraction event, this will result in a zero length edge between $\mathbf{u}$ and $\mathbf{v}$. We set the active flags of $\mathbf{u}$ and $\mathbf{v}$ to false, and replace them with a new vertex $\mathbf{u}'$. If the resulting sweep polygon is a base case, then we add a base case event to the queue. Otherwise, we calculate splitting events between $\mathbf{u}'$ and all other vertices of the sweep and insert these into the queue. We then generate contraction events for the two edges incident u'. If the event is a splitting event, we split the sweep polygon into two sweep polygons P_1 and P_2. u and v are replaced with new vertices u_1, v_1 in P_1 and u_2, v_2 in P_2. If either P_1 or P_2 we generate a base case event for it and add it to Q. Otherwise, the active flags for u and v are set to false, and new splitting events are generated and added to Q between $\mathbf{u}_1$, $\mathbf{v}_1$, $\mathbf{u}_2$, and $\mathbf{v}_2$ and the rest of the vertices in their respective split sweep polygons P_1 and P_2. New contraction events are added to the queue for the edges incident $\mathbf{u}_1$, $\mathbf{u}_2$, $\mathbf{v}_1$, and $\mathbf{v}_2$. Checking the valid flags on $\mathbf{u}$ and $\mathbf{v}$, and that $\mathbf{u}$ and $\mathbf{v}$ still lie in the same sweep polygon ensures that we only process events which are currently valid, i.e. they lie in sweep polygons which have not yet been processed. Correctness follows.

Analysis. Given the implementation details above, we prove:

Theorem 6.8.1. *Let (T, P_T) be a Lang polygon and n be the number of nodes in T. The the priority queue based implementation of the UM-skeleton algorithm computes the UM-skeleton of (T, P_T) in $O(n^2 \log n)$ time and $O(n^2)$ space.*

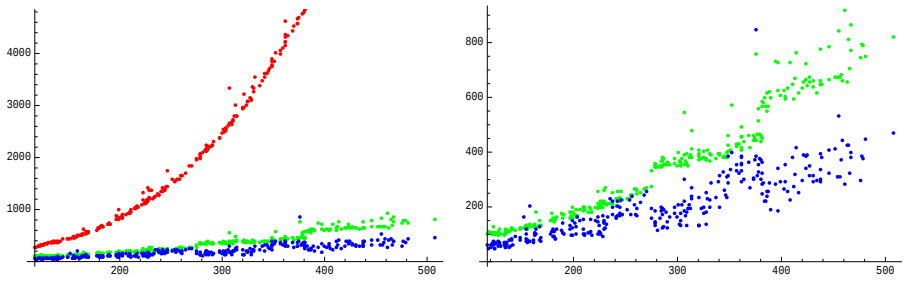
Proof. The running time of the algorithm is $O(n^2 \log n)$. When a contraction event is processed, less than $n-3$ new splitting events are added to the queue (since we gener-

ate events between $\mathbf{u}'$ and the other vertices not consecutive with it) and exactly two contraction events. When a splitting event is processed, the number of splitting events added to the queue is upper bounded by $4n$. The number of new contraction events is constant. Since there are a total of $O(n)$ events processed (Lem. 6.3.1), this means that the number of events added to the queue during the run of the algorithm is $O(n^2)$. The algorithm is initialized with $O(n^2)$ splitting events and $O(n)$ contraction events, so the total size of the queue is $O(n^2)$. Therefore, query and insertion operations on Q require $O(\log n)$ time and processing each event requires $O(n \log n)$ time. Additionally, it takes a constant time check to see if an event is valid, and thus $O(n^2 \log n)$ time to throw out all invalid events. The running time of the algorithm follows. □

6.9 Experimental Results

We implemented the naive, cyclic tournament forest (CTF), and priority queue (PQ) versions of the algorithm in Java and tested them on a MacBook Pro with a 2.9 GHz Intel Core i7 processor and 8 GB of memory. We tested each algorithm on 282 randomly generated convex polygons. Each polygon P was created by sampling the unit disk uniformly at random and computing the convex hull of the resulting point set. We then computed the straight skeleton of the convex hull. From this we extracted a tree T for which P is a Lang polygon as follows. Let (u, v) be an edge of the straight skeleton and f be one of the faces of the subdivision of the interior of P containing (u, v). Each such face f is incident to exactly one edge e of the polygon (see [4]). Project (u, v) orthogonally onto the line supporting e and take the length of the projected line segment to be the length of (u, v) in the metric tree T. It can be shown that the TSkel for (T, P) is exactly the straight skeleton of P. Finally, we randomly perturbed each leaf arc of T by adding a small uniformly chosen Δ to its length and moved the corresponding vertex of P by scaling its adjacent sides by Δ.

To run the experiment, we initialized and ran each algorithm five times on each polygon P. We measured the total time in milliseconds it took to initialize and run each algorithm and averaged the result. The results are shown in Fig. 6.7. Both the CTF and PQ based implementations perform on the same order as expected, with the PQ based implementation consistently running faster by a small factor. We believe that this performs better for two reasons: (1) the split operation on the CTF requires using the node-pointer tree representation vs. the PQ's more memory local array based representation; and (2) the current implementation does not aggressively shorten paths in the CTBs which leads to large numbers of degree 2 nodes. Better handling of these nodes should lead to a more efficient implementation.



(a) (b)

Figure 6.7: Speed comparison between the naive (red), CTF (green), and priority queue (blue) implementations. (a) shows all three while (b) shows only the CTF and priority queue based implementations. In both the x-axis is the number of points in the polygon and the y-axis is the average time in milliseconds the algorithm took to complete.

Final remark. Of the two competing implementations presented in this chapter, the one that is simpler to implement has a better behavior in practice by a constant (but not huge) factor. However, this method is (so far) oblivious of any structural properties of the universal molecule. Our second, more sophisticated approach, may perform better when combined with such structural insights. We implemented and compared these two distinct methods in the hope that the ultimate complexity of the tree-skeleton may result as a combination of ideas from both of them. A simple lower bound on computing the UM-skeleton is given by the $\Omega(n)$ size of the output.

The best-case performance of the algorithm (in the case when there are no splitting events) is also linear. However, we are ultimately interested in the universal molecule and not just the UM-skeleton. To compute the UM-skeleton, we must first add back the perpendiculars leading to a universal molecule algorithm which takes $O(n^2 \log n)$ time and $O(n^2)$ space. This is optimal in terms of space and is only a logarithmic factor from optimal in terms of time.

CHAPTER 7

EXTENDING THE UNIVERSAL MOLECULE TO NON-CONVEX CASES

Recall from Ch. 2 that Lang's TreeMaker method employs the universal molecule algorithm in its second phase to "fill in" each polygon produced by the first phase with a crease pattern. Since the universal molecule is only defined for convex polygons, this fails when the first phase of TreeMaker results in non-convex polygons. In this chapter, we remove this restriction by extending the universal molecule algorithm to all non-convex polygons produced by TreeMaker. In fact, our algorithm covers not only all possible polygons produced by TreeMaker, but also more exotic cases, like non-convex flat polygonal disks that cannot be realized in the plane without self-intersections. A key ingredient used in this section is the family of Lang surfaces we defined in Ch. 5. Recall that we put the universal molecule crease patterns into correspondence with a very restricted class of Lang surfaces. We now greatly relax these restrictions, and show how to the compute what we call the ***geodesic universal molecules***. This work is joint work with Ileana Streinu and is under review [16].

7.1 Introduction

Recall that the crease pattern computed by the universal molecule algorithm for a Lang polygon (T, P_T) has the property that it can be folded into a 3D shape which projects to 2D onto the tree T. This shape is (as we saw in Ch. 5) a flat Lang surface with convex boundary. Recall also that a Lang surface is formed using glueing operations on recursively defined intrinsic piecewise linear surfaces.

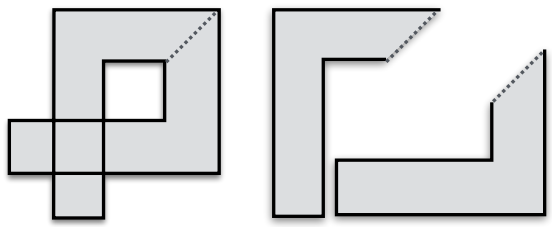


Figure 7.1: An intrinsically flat polygon that self-overlaps when (extrinsically) flattened out in the plane (left). It is intrinsically a disk, obtained by glueing two simple planar polygons along an edge (right).

The generalization we give in this chapter extends Lang's universal molecule algorithm to work with *geodesic Lang polygons* and to produce, algorithmically, a *geodesic universal molecule* crease pattern which folds into a (flat) *Lang surface* (which may not have convex boundary). A key difference between the Lang polygons define in Ch. 5 and the *geodesic* Lang polygons we define in this chapter is that the *geodesic distance* inside the (non-convex) input polygon, rather than Euclidean distance, enforces the relationship with the metric tree.

As in Ch. 5, we aim at obtaining a full characterization of the shapes (Lang surfaces) produced by this generalized algorithm, as well as of the inputs (geodesic Lang polygons) on which it works. Surprisingly, perhaps, we show that the input can be any polygon bounding a piecewise linear surface that is topologically a disk, has zero curvature, and meets certain constraints on the geodesic distance between pairs of points of the polygon that come from the tree. Such a geodesic Lang polygon should not be thought of as lying in the plane, but rather on an intrinsic surface. It is *intrinsically simple*, but an open, flat placement of a geodesic Lang polygon in the plane may self-overlap, as illustrated in Fig. 7.1.

Basic Concepts. Let T be a metric, topologically embedded tree: it has positive weights attached to each arc, and has a defined ordering or *rotation* of the incident arcs at each internal node. A polygon P_T is said to be a *doubling polygon* for T if it is metrically and combinatorially equivalent to a right-hand-turn walk around that arcs of T starting from some leaf node. We say that P_T satisfies the *geodesic Lang*

property if the geodesic distance (inside the polygon) between any two vertices of P_T is greater than or equal to the corresponding tree distance in T.

The *input* to the geodesic universal molecule algorithm described in this chapter is a tree T and *geodesic Lang polygon* P_T compatible with it. The *output* of the algorithm is a subdivision of the polygon into vertices, edges, and faces (its *crease pattern*) that is intrinsically equivalent to a *geodesic Lang surface* S_T *constructed on* T.

A *(geodesic) Lang surface* captures formally what it means, in Lang's approach, for a 3D folded origami shape to be *compatible with* and *project onto* a given metric tree. It is defined inductively from two types of building block surfaces: extrusion disks and extrusion rings. These are defined with respect to an extrusion process that embeds a *kinetic polygon* which is a doubling polygon of a *kinetic tree* (a tree in which the leaves are moving at given speeds) into a plane that sweeps upwards. The trace of the edges of the kinetic polygon define a surface which is topologically either a disk or a ring (annulus). We then define two gluing operations: one for *extending* a Lang surface by gluing it to a ring, and another for *combining* two Lang surfaces by gluing them along their boundary edges. Finally, we focus on Lang surfaces with zero-curvature at internal vertices. For the surfaces constructed in this chapter we allow the leaf nodes in a kinetic tree to move both inwards and outwards, and obtain non-convex, intrinsically simple Lang surfaces. By contrast, in Ch. 5 the leaves moved only inwards and the resulting surface was intrinsically convex.

Our main result can now be stated.

Theorem 7.1.1 (Main Theorem)**.** *Let P_T be a doubling-polygon for a tree T on a flat, disk-like piecewise-linear surface D. Then a Lang surface S constructed on T and isometric to P_T exists (and is unique) if and only if P_T is a geodesic doubling-polygon for T on D.*

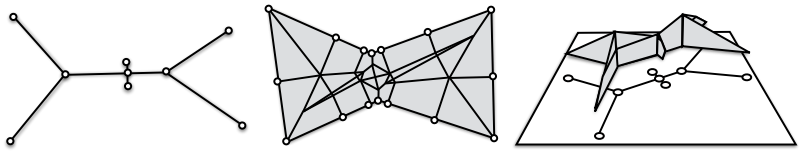


Figure 7.2: A tree (left), geodesic Lang polygon subdivided by its geodesic universal molecule (middle), and an intrinsically equivalent Lang surface (right).

Overview. The proof of the theorem is broken into three parts. In Sec. 7.5 we prove the necessity of the geodesic Lang property on the Lang surface. The second part is the sufficiency of the property, which is proven by describing an algorithm that takes as input a geodesic Lang polygon and produces as output a universal molecule (Sec. 7.6). See Fig. 7.2. The third is the uniqueness claim, proven in Sec. 7.7.2. Precise definitions are given in Sec. 7.2, and a few useful properties of geodesic Lang polygons are investigated in Sec. 7.3.

7.2 Concepts

We recall some of the basic concepts from Ch. 2 needed for this chapter.

7.2.1 Piecewise linear metric surfaces

Our primary objects of interest, both Lang polygons and Lang surfaces, are ***piecewise linear metric surfaces*** (hence ***surfaces***) obtained by gluing flat, polygonal faces together along whole edges.

Realizations of piecewise linear surfaces. A ***realization*** of a surface is a map taking each vertex to a point in $\mathbf{R}^3$, each edge to a straight-line segment, and each face to a flat polygon in $\mathbf{R}^3$ such that the edges and faces maintain their size and shape. We say two surfaces are ***equivalent*** if one is a realization of other. Notice that this is a slightly stronger version of equivalency than simply isometry, since an isometry between surfaces does not require that the surfaces have the same subdivi-

sion into vertices, edges, and faces. Here, however, we do not consider two different subdivisions of the same surface to be equivalent.

Intrinsic vs. extrinsic properties Properties of a surface that are true in any realization are ***intrinsic***, while those that depend on a particular realization are ***extrinsic***. This distinction is particularly important for our purposes, because two different foldings of the same origami crease pattern are intrinsically the same surface but differ in their extrinsic properties (such as the dihedral or "folding" angle between faces). Showing that a surface is a folding of another amounts to showing that the two surfaces only differ extrinsically. Important intrinsic properties include the surface's:

- *topology*, which in this chapter is either a disk (***disk-like***) or an annulus (***ring-like***); since these are the only two topologies we consider, each edge is either incident to exactly one face (a ***boundary edge***), or to two faces (an ***interior edge***);
- *(intrinsic) curvature* of the surface at a vertex (defined in the next paragraph); and
- the *geodesic distance* between two points on the surface (defined shortly).

An example of an extrinsic property is the *dihedral angle* between two faces at an edge.

Curvature. Since our surfaces are piecewise linear, the curvature is concentrated at the vertices. A vertex has a ***face angle*** in each of its incident faces, and its ***angle sum*** is the sum over all its face angles. The (intrinsic Gaussian) ***curvature*** at a vertex is given by 2π minus its angle sum. If every internal vertex of a surface has zero curvature then the surface is (intrinsically) ***flat***, which does not require that it be realized in a single plane. A realization of a flat surface in which the dihedral angles at all interior edges is π is an ***open, flat realization***. In these terms, both the initial crease pattern drawn on the paper and the final folding of the origami are

(intrinsically) flat, but only the first is in an open, flat realization. If for a given surface there exists an open, flat realization, then we say that the surface is ***flattenable***. Flattenability implies that the surface is (intrinsically) flat. The converse is true for all disk-like surfaces, but not for all ring-like surfaces. For instance, if one removes the top and bottom face from a cube, the resulting ring-like surface is flat (its curvature is zero everywhere), but it is not flattenable, since it has no open, flat realization.

Geodesic distances and visible pairs Given a surface S, the ***geodesic distance*** between two points p and q, denoted $d_S(p, q)$, is the length of the shortest path between them, called the ***geodesic path***. On a piecewise linear surface, this is a polygonal chain and if the surface is a disk, is unique. If the geodesic path between two points p and q is (intrinsically) straight we say that (p, q) is a ***visible pair***. Note that the geodesic distance $d_S(p, q)$ satisfies the usual triangle inequality–for all p, q, r, $d_S(p, q) \leq d_S(p, r) + d_S(r, q)$.

Polygons. Thus far in this dissertation, we have used the term "polygon" in what might be called its common usage–to refer to polygons drawn in the plane. These are typically defined by giving an ordered cycle of points, and the polygon is the point set together with the straight line segments between consecutive points.

Alternatively, such a polygon can be defined as an ordered cycle of line segments drawn in the plane end-to-end. This second definition generalizes well to piecewise-linear surfaces. Let S be a piecewise-linear surface. We call an intrinsically straight path between two points $\mathbf{p}$ and $\mathbf{q}$ of S a ***geodesic segment***, denoted $\mathbf{pq}$. Note here that a geodesic path of a surface may not be straight (for instance on the interior of a planar non-convex polygon, a geodesic path may bend around a reflex vertex), but we require that a geodesic segment be straight. A (geodesic) ***polygon*** on S is a cycle of geodesic segments on S that are connected end-to-end (to make this precise, we order each segment $\mathbf{pq}$ by giving it a source vertex $\mathbf{p}$ and a destination vertex $\mathbf{q}$; by

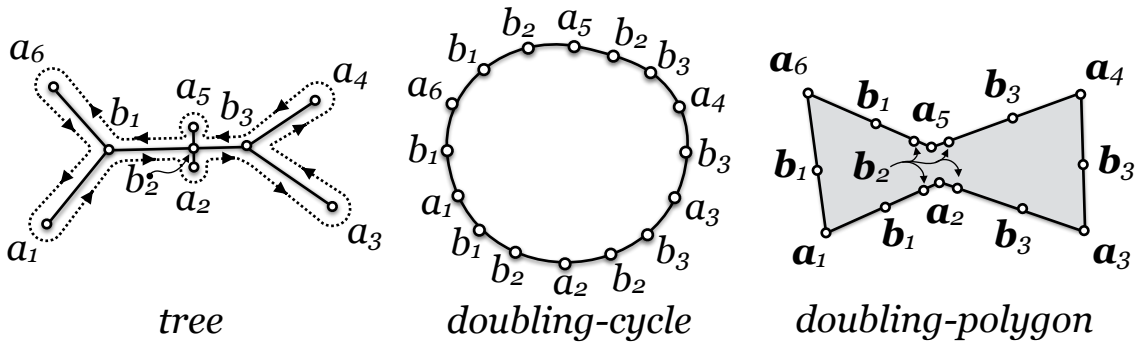


Figure 7.3: A tree, doubling cycle, and doubling-polygon.

"end-to-end" we mean that for two consecutive, the destination point of the first is the source point of the second).

When a polygon is simple, i.e. it does not self-touch or self-cross on the intrinsic surface, then we view it together with the piece of the surface that it bounds; in other words, a simple polygon on S is itself a disk-like surface. We remark that if we flatten out S into the plane, the polygon may self-overlap even though it is simple on S (Fig. 7.1). A vertex of a polygon with angle sum less than π is said to be ***convex***, equal to π a ***marker***, and greater than π ***reflex***.

7.2.2 Metric trees, metric doubling cycles, and doubling polygons

A ***metric tree*** (T, w) is a tree T and a weight function w that maps each arc[1] of T to a positive weight or ***length***. We assume that a cyclic ordering, or rotation, is given for the incident arcs at each node[2]. The ***metric doubling cycle*** for T is the pair (C_T, w) where C_T is the cycle given by starting at any leaf node and listing the nodes encountered by walking around T while respecting the ordering of incident arcs and w maps each edge of C_T to the length of its corresponding tree arc. See Fig. 7.1. In such a walk, each edge is traversed once in each direction, and each vertex is visited

[1]To avoid confusion, we use the terms ***node*** and ***arc*** to refer to the elements of a tree, and ***vertex*** and ***edge*** to refer to the elements of a polygon or embedded straight-line graph.

[2]Such a tree is sometimes called a ribbon tree, or a topologically embedded tree.

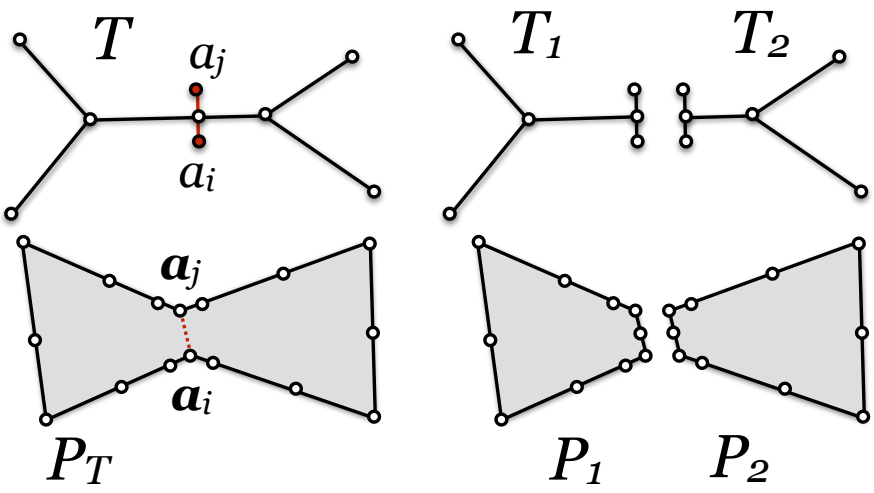


Figure 7.4: Splitting a tree and corresponding doubling polygon between a_i and a_j.

the number of times equal to its degree. A ***doubling polygon*** P_T is a polygon that is combinatorially and metrically a doubling cycle for T.

Notation. It is convenient to separate the n leaf nodes and m internal nodes of a tree T into two sets $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$, respectively. In order to make clear the correspondences between a tree T and a doubling polygon P_T, we use **bold face** to denote vertices of the polygon and *italics* to denote corresponding nodes in the tree. For instance, the vertex $\mathbf{a}$ in P_T corresponds to the leaf node a in T and the edge $\mathbf{ab}$ corresponds to the leaf arc ab.

Splitting trees, cycles, and polygons Given a tree T and two leaf nodes a_i and a_j, the ***splitting operation*** returns two trees T_1 and T_2 corresponding to the part of the tree to the left of (and including) the path from a_i to a_j in T, and the part to the right (resp). To split a doubling cycle C_T between a_i and a_j, we first split C_T into two open chains C_1 and C_2, one from a_i to a_j and the other from a_j back to a_i. We then close each chain using a copy of the path from a_i to a_j in T. The chains C_1 and C_2 are then doubling cycles for T_1 and T_2 (resp). In a doubling polygon P_T we allow this operation only if $(\mathbf{a}_i, \mathbf{a}_j)$ is a visible pair, and the geodesic distance from $\mathbf{a}_i$ to $\mathbf{a}_j$ is equal to $d_T(a_i, a_j)$. The split in the polygon is performed by introducing

a ***splitting edge*** along the shortest path between $\mathbf{a}_i$ and $\mathbf{a}_j$ and subdividing it into edges so that it is metrically and combinatorially equivalent to the path between a_i and a_j in the tree. See Fig. 7.4.

7.3 Geodesic Lang polygons

The first object of interest is the geodesic Lang polygon. This is a generalization of the Lang polygons we defined in Ch. 5. Let S be a piecewise linear zero-curvature disk-like surface and let T be a metric tree with strictly positive edge weights (no degeneracies). We consider a doubling polygon P_T for T on a surface S which may be self-touching but not self-crossing, and thus has a well-defined *interior* on the surface. If the polygon is not self-touching, the interior is itself a disk-like surface; otherwise, it may have several disk-like components. A *geodesic* is the shortest path between two points on the polygonal boundary, lying entirely in the closure of the interior of the polygon.

We say that a doubling polygon T_T satisfies the ***geodesic Lang property*** on S if for all pairs of points $(\mathbf{u}, \mathbf{v})$ on the boundary of P_T, their geodesic distance is greater than the corresponding tree distance, i.e. $d_S(\mathbf{u}, \mathbf{v}) \geq d_T(u, v)$. If further, each vertex of P_T corresponding to an internal node of T is a marker (interior angle is π), then it is a ***geodesic Lang polygon***.

Negative boundary curvature. In general we require that the interior angle measure at each vertex of P_T be less than 2π; however, we allow higher angle measures by the following construction. Suppose that we two pairs (T, P_T) and $(T', P'_{T'})$ of trees and doubling polygons such that there is a side $\mathbf{a}_i\mathbf{a}_j$ (in ccw order) in P_T that is *equivalent* to a side $\mathbf{a}'_j\mathbf{a}'_i$ in $P'_{T'}$. By 'equivalent' we mean that the path from a_i to a_j in T has the same number of arcs (with the same lengths) as the path from a'_i to a'_j in T'. We then construct a new doubling polygon (T'', P''_T) by gluing T to T' the equivalent paths in the tree, and by gluing P_T to P'_T by identifying the sides $\mathbf{a}_i\mathbf{a}_j$ and

$\mathbf{a}'_i\mathbf{a}'_j$. This is the inverse operation to the splitting of a tree and polygon depicted in Fig. 7.4. The interior angle at the vertex $\mathbf{a}''_i$ in $P''_{T''}$ is the sum of the interior angles of $\mathbf{a}_i$ and $\mathbf{a}'_i$ in P_T and $P'_{T'}$. This allows us to arbitrarily increase the interior angle sum at a vertex so long as the property above is satisfies. We call such a doubling polygon ***well-constructed*** and allow that a geodesic Lang polygon have negative on its boundary only in the case that it is well-constructed. Let us now investigate two important properties of geodesic Lang polygons.

The geodesic Lang property on visible pairs implies the geodesic Lang property on all pairs. We now derive a key property of geodesic Lang polygons. We show that for a well-constructed doubling polygon to be a Lang polygon, it is sufficient that it satisfies the Lang property for all visible pairs. This is used in Sec. 7.6 where we describe an algorithm for computing crease patterns on a geodesic Lang polygon that relies on this property.

Lemma 7.3.1. *Let P_T be a well-constructed doubling polygon for T on S such that each vertex $\mathbf{b}$ of P_T corresponding to an internal node b of T is a marker. If the geodesic Lang property holds for all pairs of visible corners on P_T, then (T, P_T) is a Lang polygon.*

Proof. We prove the contrapositive–that a violation of the geodesic Lang property for a non-visible pair implies a violation for a visible pair. Let $(\mathbf{a}_i, \mathbf{a}_j)$ be a non-visible pair for which the geodesic Lang property does not hold. Assume for contradiction, that the geodesic Lang property holds for all visible pairs. Let p denote the geodesic path from $\mathbf{a}_i$ to $\mathbf{a}_j$ in P_T. The path p is a polygonal path and each of its interior vertices is a corner of the boundary of P_T. Denote the vertices along p in order by $\mathbf{A}_0 = \mathbf{a}_i, \mathbf{A}_1, \ldots, \mathbf{A}_k = \mathbf{a}_j$. Each consecutive pair $(\mathbf{A}_m, \mathbf{A}_{m+1})$ is a visible pair and since each $\mathbf{A}_m$ is a corner in P_T, it corresponds to a leaf node A_i in T. Then by hypothesis, for each m from 1 to $k-1$ we have $d_{P_T}(\mathbf{A}_m, \mathbf{A}_{m+1}) \geq d_T(A_m, A_{m+1})$. The

length of p is given by $\sum_{m=0}^{k-1} d_{P_T}(\mathbf{A}_m, \mathbf{A}_{m+1})$. Thus we have $\sum_{m=0}^{k-1} d_{P_T}(\mathbf{A}_m, \mathbf{A}_{m+1}) \geq \sum_{m=0}^{k-1} d_T(A_m, A_{m+1}) > d_T(A_1, A_k)$ (the last inequality follows from the triangle inequality on T). Therefore the pair $(\mathbf{a}_i, \mathbf{a}_j)$ satisfies the geodesic Lang property, a contradiction. □

TreeMaker always produces geodesic Lang polygons. We now observe that the polygons in any solution to the optimization problem solved in the first phase of TreeMaker are geodesic Lang polygons. This guarantees that when the first phase of TreeMaker produces an output, the polygons that it produces, when not convex, satisfy the preconditions required by the geodesic universal molecule algorithm of Section 7.6 to work.

Lemma 7.3.2. *Let P_T be a simple, possibly non-convex planar doubling polygon for a metric tree T, satisfying the Lang property that the Euclidean distance between vertices exceeds the tree distance. Then P_T is a geodesic Lang polygon for T.*

7.4 Generalized sweep of a geodesic Lang polygon

We now define a process on a Lang polygon called a *generalized sweep*. This concept is used in several places in the remainder of the chapter. In Sec. 7.5, we show how to construct a particular family of surfaces we call *Lang surfaces* by an extrusion process. The boundary of each Lang surface is shown to be a geodesic Lang polygon and we end the section by showing that *the extrusion process is equivalent to a generalized sweep starting from its boundary.* Next, in Sec. 7.6, we give our generalization of the universal molecule algorithm to geodesic Lang polygons, which *uses a generalized sweep on the interior of its input Lang polygon* to generate a crease pattern. These crease patterns are shown (in Sec. 7.7.1) to be equivalent to Lang surfaces. Finally, in Sec. 7.7.2 we use the concept of a generalized sweep to prove the uniqueness claim of

Theorem 7.1.1 by showing, essentially, that any Lang polygon gives rise to exactly one generalized sweep. (How this implies the uniqueness claim is detailed in Sec. 7.7.2.)

We now turn to the definition of the generalized sweep of a geodesic Lang polygon. Recall that a Lang polygon is a pair (T, P_T) of a tree T and a polygon P_T which is drawn on some underlying surface (we restrict ourselves here to surfaces of zero-curvature). To define a generalized sweep we need two additional processes–a *kinetic stretching process* defined on the tree in which the leaves shrink and a *parallel sweep process* in the polygon by which its edges are moved inwards in parallel at unit speed. We define these in Sec. 7.4.1 and end by defining the generalized sweep in Sec. 7.4.2.

7.4.1 Kinetic trees and parallel sweeps

Kinetic trees. We make a metric tree (T, w) ***kinetic*** by attaching a ***stretching speed*** $s(ab)$ to each leaf arc ab. The length of an arc ab at time $t \geq 0$ is given by $w(ab) + t\, s(ab)$. This gives rise to a family of trees $T(t)$ parametrized by t. When the tree is embedded, we extend this motion to the embedded tree by moving leaf nodes inwards or outwards along the supporting line of the leaf arc. If $s(ab)$ is positive, then we say the arc is ***growing***, otherwise ***shrinking***[3]. This is naturally extended to any doubling cycle C_T for T to form a family of doubling cycles $C_T(t)$–simply grow/shrink each edge of C_T at the same speed as its corresponding arc in T. This trivially maintains the property that at each time t, the cycle $C_T(t)$ is a metric doubling cycle for $T(t)$.

Parallel sweep of a polygon. A ***parallel sweep*** of a polygon is given by moving the edges of the polygon inwards at unit speed in such a way that each edge remains parallel to its initial position. Each edge grows or shrinks to maintain incidence with its adjacent edges.

---

[3]Note that in Ch. 5, we only allowed a leaf arc to shrink. Here we must allow both shrinking and growing to maintain certain correspondences with parallel sweeps of non-convex polygons.

7.4.2 The generalized sweep

Overview. Our goal in this section is to define a generalized sweep for a Lang polygon (T, P_T). The basic idea is that we make the tree kinetic and grow/shrink its leaf arcs while simultaneously performing a parallel sweep of the polygon. At certain points we may split the tree and polygon (according to the splitting operation defined in Sec. 7.2). Thus, at any given point we may have multiple shrinking polygon and tree pairs (hence the term "generalized"). Ultimately, we define this process so that the pair of the kinetic tree (with leaf arcs that are growing or shrinking) and the parallel polygon (with edges that are growing or shrinking) maintain the geodesic Lang property–throughout this process the pair forms a Lang polygon (meaning that the sweeping polygon is a doubling polygon for the growing/shrinking tree and the geodesic Lang property is satisfied). Maintaining this invariant requires that we process two types of events that occur in the sweep: contraction events and splitting events.

Contraction events. The first event type occurs when an arc of the tree and its corresponding edges in the polygon shrink to zero-length. Combinatorially, the zero-length arc in the tree is removed and the zero-length edges in the sweeping polygon are replaced with a single vertex.

Splitting events. The second event type occurs when the geodesic Lang property is satisfied with equality for some non-consecutive visible pair of corner vertices $(\mathbf{a}_i, \mathbf{a}_j)$ in P_T. We call this a ***potential splitting event*** because at this point the splitting operation may be applied to the tree and polygon. A potential splitting event occurs because the rate at which the distance is changing for some pair $(\mathbf{a}_i, \mathbf{a}_j)$ in the sweeping polygon is not necessarily the same as the rate at which the corresponding distance between a_i and a_j is changing in the tree. Thus, a pair that satisfies the geodesic Lang property initially with *inequality* may satisfy the geodesic Lang prop-

erty with *equality* at some future time. In this case *we allow* that a splitting operation be applied to the polygon for $(\mathbf{a}_i, \mathbf{a}_j)$ and to the tree for (a_i, a_j) to obtain geodesic Lang polygons (T_L, P_L) and (T_R, P_R). We then continue the sweep independently in each. Note that it is not obviously the case that we *must split* at such an event in order to maintain the geodesic Lang property. For instance, it may be that immediately after such an event the distance between the two vertices in the sweeping polygon increases more quickly than the distance between the corresponding tree nodes. In such a case, even though the geodesic Lang property holds with equality, we do not need to split in order to maintain the geodesic Lang property and thus we can choose either to split or not. On the other hand, if immediately after such an event the geodesic Lang property is violated, then we are *forced to split* in order to maintain the geodesic Lang property. We call a potential splitting event at which the splitting operation is actually applied simply a ***splitting event***. In the special case of negative boundary curvature (see Sec. 7.3) we require that the sweep be split immediately so that each resulting sweep polygon does not have negative boundary curvature. This requirement comes from the fact that a parallel offset polygon is only well-defined for polygons without negative boundary curvature.

A generalized sweep. Let (T, P_T) be a Lang polygon. Make T kinetic by assigning to each leaf arc ab of T a speed of $-1/\tan(\theta_{\mathbf{a}})$ where $\theta_{\mathbf{a}}$ is half the interior angle measure at $\mathbf{a}$. This speed is not arbitrary, but is chosen so that the speed at which leaf arc shrinks is the same as the speed at which its corresponding edges in the polygon shrink (this follows from elementary trigonometry). Given these speeds, then, sweeping the polygon and shrinking the tree as described above maintains that the polygon is a doubling cycle for the tree throughout the process. Now suppose we perform a sweep of the polygon and stretching of the tree as described above in which we process all contraction events, and optionally split the polygon and tree at some of the splitting events. If, throughout the process, we maintain the geodesic

Lang property, then we call this process a ***generalized sweep***. Note here that this definition allows that there may be multiple possible generalized sweeps for the same polygon and tree depending on which whether we actually split at potential splitting events. Recall that this occurs because we allow that the sweep *not be split* at a potential splitting event so long as not doing so does not violate the geodesic Lang property; however, we will see in Sec. 7.7.2 that in order to maintain the geodesic Lang property we must *always split* at potential splitting events. In other words, any time the sweep arrives at a potential splitting event, to continue past the event without actually splitting causes the sweeping polygon and tree to violate the Lang property, and thus the geodesic Lang property fails to hold. Ultimately, we will see that this implies that there is exactly one sweep for a given geodesic Lang polygon. This is used in Sec. 7.7.2 to prove the uniqueness claim from Theorem 7.1.1.

Can a generalized sweep self-touch? One important property of a generalized sweep is that the sweeping polygon never self touches. This is not the case in the related parallel sweep used in the definition of the straight skeleton of a non-convex polygon [4]. Were such an event to occur, we would need one entirely different type of "splitting event" in which the sweeping polygon is split at the point at which the polygon self-touched, as is the case for the straight skeleton of a non-convex polygon. We now show that in a generalized sweep it is not possible to reach such an event. To prove this we show that if such an event occurs at some point during the sweep, then the sweep violates the geodesic Lang property, and thus is not a generalized sweep. Recall that initially we have one sweeping polygon and stretching tree, but the polygon and tree may split at splitting events so that at any given time during a generalized sweep, we have a collection of sweeping Lang polygons. Observe that one of the sweeping polygons cannot touch another sweeping polygon since each sweeping polygon always moves its edges towards its interior. Thus, as soon as a polygon is split, the resulting two sweep polygons diverge. What we are interested, then, is whether

one of these parallel sweep polygons can self-touch, meaning that either a vertex of the polygon "hits" some edge elsewhere in the polygon or two (or more) vertices "hit" each other. We now show that in a generalized sweep, this is not the case:

Lemma 7.4.1. *At no point during a generalized sweep does a parallel sweep polygon self-touch.*

Proof. Assume not. Suppose for contradiction that at some time t, a parallel sweep polygon self touches. Take the minimum time t at which this occurs. Denote the polygon by P_T and its corresponding tree by T. There are two cases. In the first case, some vertex of the polygon touches an edge elsewhere in the polygon. In this case, the collision vertex must be reflex. Otherwise, the polygon would already not be a simple, but this implies that t is not the minimum time at which such an event occurs since we start with a simple polygon, and splitting events always produce simple polygons. In the second case, two vertices touch each other simultaneously but are not part of the same contraction event. Again this implies that at least one of the vertices is reflex. Otherwise, since the sweep always moves towards the interior of the polygon the edges incident to the two vertices will have crossed immediately prior to the event, again contradicting that t is the minimum time of such an event. We now prove that in each of the two cases we arrive at a contradiction:

Case 1: Suppose vertex $\mathbf{a}_j$ hits edge $\mathbf{a}_i\mathbf{a}_{i+1}$ in the polygon. That $\mathbf{a}_j$ hits some edge in the sweep implies that $\mathbf{a}_j$ is reflex in P_T, and thus the corresponding leaf arc is growing. By definition, $\mathbf{a}_i\mathbf{a}_{i+1}$ corresponds to a path in T between leaf nodes a_i and a_{i+1} and $\mathbf{a}_j$ corresponds to the leaf node a_j in T. Since a_j is a leaf node and the corresponding leaf arc is growing, a_j does not lie on the path between a_i and a_{i+1} in T. Therefore we have that $d_T(a_i, a_{i+1}) < d_T(a_i, a_j) + d_T(a_j, a_{i+1})$. Since $(\mathbf{a}_i, \mathbf{a}_{i+1})$ is an edge, then $d_{P_T}(\mathbf{a}_i, \mathbf{a}_{i+1}) = d_T(a_i, a_{i+1})$, and thus we have that $d_{P_T}(\mathbf{a}_i, \mathbf{a}_j) + d_{P_T}(\mathbf{a}_j, \mathbf{a}_{i+1}) < d_T(a_i, a_j) + d_T(a_j, a_{i+1})$, which entails that either $d_{P_T}(\mathbf{a}_i, \mathbf{a}_j) < d_T(a_i, a_j)$ or $d_{P_T}(\mathbf{a}_j, \mathbf{a}_{i+1})$ contradicting the geodesic Lang property on P_T.

Case 2: Let $\mathbf{a}_i$ and $\mathbf{a}_j$ be the touching vertices that are not part of the same contraction event and without loss of generality suppose $\mathbf{a}_i$ is reflex in P_T. Then the leaf arc incident to a_i has positive, non-zero length (since it is growing), and so a_i is at least some positive distance from any other node in T. Since $\mathbf{a}_i$ and $\mathbf{a}_j$ are not part of the same contraction event, then a_i and a_j are distinct in T and thus $d_T(a_i, a_j) > 0$. But the distance from $\mathbf{a}_i$ to $\mathbf{a}_j$ is zero, hence the geodesic Lang property is violated. □

As a consequence we have:

Lemma 7.4.2. *There exists a generalized sweep for any given geodesic Lang polygon (T, P_T) (drawn on some flat surface D).*

Proof. Make T kinetic as in the definition of a generalized sweep and perform a parallel sweep of P_T and simultaneous stretching of T. We will show that the following generalized sweep exists–whenever we encounter a potential splitting event we split. If multiple potential splitting events occur simultaneously, then we take one after another, splitting until no potential splitting events remain, and then continue. Note here that we leave open the possibility that "processing" one potential splitting event removes another by separating its two vertices on opposite sides of the split (although we will see in Sec. 7.7.2 that this is not possible).

The sweep moves the sides of P_T inwards towards its interior, and so as the sweep progresses, we must either encounter (1) a contraction event, (2) a potential splitting event, or (3) an event in which the sweeping polygon self-touches. By definition, the sweep maintains that for consecutive pairs $\mathbf{a}_i$ and $\mathbf{a}_{i+1}$ the distance in the tree and the distance in the sweeping polygon is equal (i.e. $d_T(a_i, a_{i+1}) = d(\mathbf{a}_i, \mathbf{a}_{i+1})$). From this we can rule out case (3). Assume we get to a point at which P_T self touches. Then by Lemma 7.4.1 (T, P_T) is no longer a geodesic Lang polygon. But this means that at some earlier time, there must have been a potential splitting event at which we did not split, a contradiction.

Thus, as long as we take as our rule that we always split at potential splitting events, then we will encounter events of type (1) and (2) only. But each contraction event removes at least one vertex from the sweeping polygon (and one leaf from the tree), and each splitting event splits the polygon and tree into two polygons and two trees each with strictly fewer vertices/leaf nodes and at least three vertices/leaf nodes (since splitting events always occur for non-consecutive vertices).

The result then follows by induction on the number of events encountered in the sweep. □

7.5 Lang Surfaces

We now review the Lang surfaces, which were defined in Ch. 5. We now investigate the set of zero-curvature Lang surfaces in full generality. Recall that Lang surfaces are built with respect to a tree T and are formed by combining a small set of basic building block elements, according to certain gluing rules, to form disk-like surfaces.

As before, to define the building blocks, we first make the tree T *kinetic* by allowing its leaf arcs to grow or shrink. The building blocks are then formed via an extrusion process. We first define kinetic trees and their counterparts kinetic doubling cycles and kinetic doubling polygons in Sec. 7.4.1. We then briefly review our construction of Lang surfaces in Sec. 7.5.1. We define two families of building block surfaces and two operations, extension and combination, for gluing them together to form Lang surfaces. This is the same construction as in Ch. 5, *except that we allow tree edges to both grow and shrink, and we remove the restriction that a Lang surface have a convex boundary.* In 7.5.2, we investigate the properties of Lang surfaces with zero curvature and show that the boundary polygon P_T of a zero-curvature Lang surface S constructed on a tree T is a geodesic Lang polygon, proving the necessity of the geodesic Lang property in Theorem 7.1.1.

We end this section by observing that the extrusion processes for creating the building blocks of a Lang surface can be chained together to form an intrinsic ***parallel sweep*** of the surface with splitting events. A parallel sweep of the surface sweeps the boundary of the surface inward towards in such a way that each edge of the sweep remains (intrinsically) parallel to its original position and all edges move at unit speed. A splitting event splits the sweeping polygon into two polygons and the sweep continues recursively in each. We used this fact in Ch. 5 to put the universal molecules into correspondence with the zero-curvature Lang surfaces with convex boundary. We observe that concepts from the convex case transfer to the general case studied in this chapter because the parallel sweep of a Lang surface (convex or otherwise) locally proceeds in the same way as in the convex case in the plane. As in the convex case, in general a Lang surface may have non-zero curvature at its interior vertices, but in the end we will impose zero-curvature on all interior vertices. This, however, explicitly allows for high curvature (angle sum $> 2\pi$) on the boundary, which presents a potential problem–what does the sweep look like locally at such a vertex? In our construction, however, these high curvature vertices only occur at combination operations, which correspond to a splitting of the sweep polygon. After the split each vertex of the sweeping polygon has angle less than 2π, and so the sweep looks locally like a sweep of a polygon in the plane.

7.5.1 Constructing Lang surfaces

To define Lang surfaces, we first define two types of building blocks, extrusion disks and extrusion rings. Each is built with respect to a kinetic tree via an extrusion process. The boundary polygon(s) for an extrusion disk or ring are doubling cycles. We then give two gluing operations, extension and combination, for joining them. The gluing operations can be applied only if the two input surfaces meet certain conditions coming from the tree. In Ch. 5, we restricted all leaf arcs to shrink and the

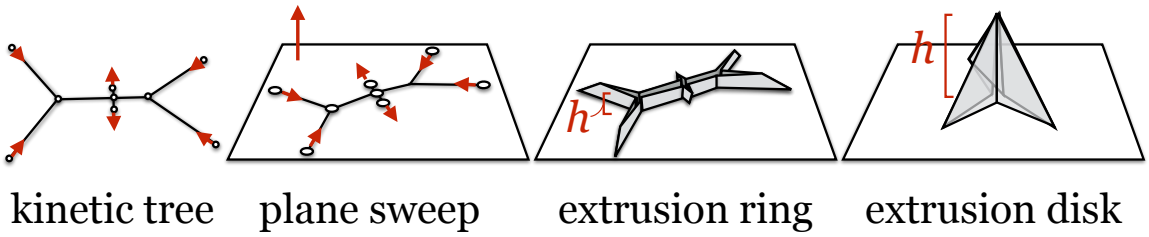


Figure 7.5: A kinetic tree, the kinetic tree embedded in a sweeping plane, an extrusion ring of height h and an extrusion disk of height h. Note that we "open up" the ring and disk slightly for illustration purposes.

combination operation to produce only surfaces with convex boundary. This ensures that the resulting Lang surfaces are convex. In this chapter, however, we fully generalize to Lang surfaces with non-convex boundary by allowing leaf arcs to grow as well as shrink and by allowing the combination operation to result in surfaces with (intrinsically) non-convex boundary polygons. That said, many of the properties of Lang surfaces studied in Ch. 5 are preserved; we point them out along the way.

Extrusion surfaces. Recall that the two types of building block surfaces are defined with respect to a kinetic tree (T, w, s) and a positive ***extrusion height*** h. The extrusion process is given by the following construction. First, we fix an embedding of the tree in the xy-plane. Next, we shrink or grow each leaf arc (according to its speed $s(ab)$) while simultaneously moving the plane containing the tree upwards in the positive z-direction at unit speed. We simulate both of these motions for t from 0 to h. At time t we have $T(t)$ embedded in the $z = t$ plane. Next, we obtain a doubling polygon $P_T(t)$ for $T(t)$ by embedding the doubling cycle $C_T(t)$ directly "on top of" $T(t)$ in the sweeping plane, meaning each vertex $\mathbf{v}$ of $P_T(t)$ is placed directly on top of its corresponding node v in the tree $T(t)$ in the $z = t$ plane. We call this polygon $P_T(t)$ (embedded in the $z = t$ plane) the ***extrusion polygon*** at height t. The ***extrusion surface*** of height h for (T, w, s) is the trace of the edges of P_T in this sweep. See Fig. 7.5. We restrict h to be not greater than the first time t at which an arc shrinks to zero length in $T(t)$.

Extrusion disks. If we assign each arc ab of T a stretching speed $s(ab)$ equal to $-h/d_T(a,b)$ and T has a single internal node, then all of the edges of the extrusion polygon shrink to a single point $\mathbf{p}$ at $t = h$. The resulting extrusion surface is topologically a disk, which we call it an ***extrusion disk***. The point $\mathbf{p}$ is the single interior vertex for the disk. Its curvature is $2\pi - 2\sum_{ab\in T} \arctan(d_T(a,b)/h)$, and there is a unique height h for each tree T resulting in an extrusion disk of zero curvature. Each face is a right triangle with one edge equal to the initial doubling polygon edge in the xy-plane, one edge equal to an edge of length h lying perpendicularly above the xy-plane, and the remaining edge a hypotenuse traced by a vertex of the extrusion polygon corresponding to a leaf node. There is a degenerate situation that results in a disk. If a tree has all of its leaf arcs incident to only two different internal nodes, then the tree $T(h)$ is a path (has exactly two leaf nodes), and we make the surface a disk by identifying the edges of the corresponding edges of the extrusion polygon at height h. Handling this second case is a straightforward extension of the first, and so we focus only on the first.

Extrusion rings. The second type of extrusion surface is defined for a kinetic tree such that no leaf arc shrinks to zero-length at a time $t < h$ and the tree $T(h)$ has at least three leaf nodes. The resulting ***extrusion ring*** of height h is a ring-like surface with lower and upper boundary polygons that are doubling polygons of T and $T(h)$. See Fig. 7.5. We note that because each vertex of an extrusion ring is on the boundary, all extrusion rings are flat, though a given extrusion ring may not have an open, flat realization (see Sec. 7.2.1).

Boundary curvature and face geometry of extrusion surfaces. Recall that the boundary polygon of an extrusion surface is a doubling polygon for the tree T. Each vertex $\mathbf{v}$ of the boundary is incident to exactly two faces in the surface. If the vertex corresponds to an internal node of T, then its (x,y)-coordinates do not change throughout the extrusion process. This implies that the two face angles incident to $\mathbf{v}$

are each $\pi/2$ and the angle sum is π. This also implies that an edge in the extruding polygon corresponding to an internal arc in T traces out a rectangular face. On the other hand, a leaf node moves so that its incident leaf arc grows or shrinks according to the speed $s(ab)$. An edge of the extruding doubling polygon corresponding to a leaf arc in T traces out a right triangle if the arc shrinks to zero at $t = h$, or a right trapezoid otherwise. An edge corresponding to an internal arc of the tree traces out a rectangular face. In particular, this means that the vertices of the boundary of an extrusion disk or lower boundary of an extrusion ring have angle sum less than 2π, or rather the curvature at these vertices is non-negative.

Operations for constructing Lang surfaces. Lang surfaces are obtained by starting with extrusion disks and rings and combining them using the following two operations, *combination* and *extension*. A Lang surface constructed on an embedded kinetic metric tree (T, w, s) is a disk-like piecewise linear surface in $\mathbf{R}^3$ whose boundary is a doubling polygon T. Lang surfaces are defined inductively. All extrusion disks are Lang surfaces. New Lang surfaces are formed by either applying the extension operation to a Lang surface and an extrusion ring, or by applying the combination operation to two Lang surfaces (in each case meeting certain preconditions).

The extension operation. This operations takes as input an extrusion ring R of height h and a Lang surface S with the precondition that the upper boundary polygon of R and the boundary polygon of S are the same doubling polygon for the same tree (except that the upper boundary polygon of R is in the $z = h$ plane and the boundary of S is in the xy-plane). We ***extend*** S with R by translating S upwards in the positive z-direction by h, bringing its boundary polygon into the $z = h$ plane. We then identify the corresponding edges of the upper boundary polygon of R and the boundary polygon of S to form the output Lang surface. See Fig. 7.6.

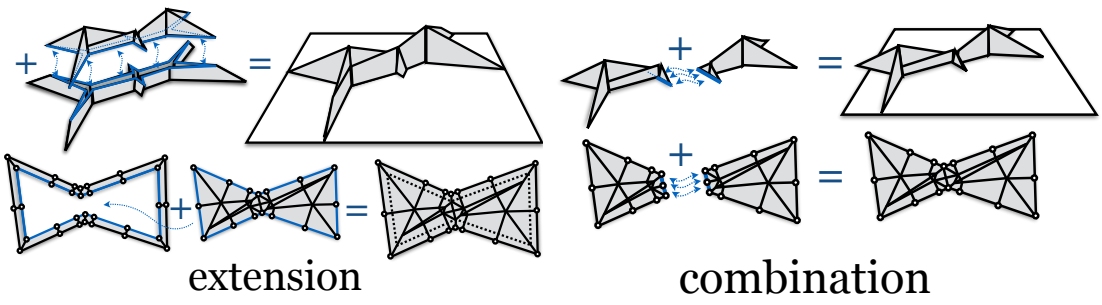


Figure 7.6: The extension (left) and combination (right) operations depicted extrinsically (top) and intrinsically (bottom).

The combination operation. This operation takes as input two Lang surfaces S_1 and S_2 constructed on trees T_1 and T_2 (resp.). The precondition for this operation is that there exists a tree T and a pair of leaf nodes (a_i, a_j) in T, such that when T is split between a_i and a_j, the result is T_1 and T_2. In particular, this means that a_i and a_j are consecutive leaf nodes in T_1 and T_2 and thus appear as consecutive corner vertices $\mathbf{a}_i$ and $\mathbf{a}_j$ in the boundary polygons of both S_1 and S_2. The paths between $\mathbf{a}_i$ and $\mathbf{a}_j$ in both S_1 and S_2 correspond to the path in T between a_i and a_j. We ***combine*** S_1 to S_2 along this ***gluing path*** by identifying the corresponding edges along the path. Note that because a_i and a_j are consecutive in S_1 and S_2, then this gluing path is intrinsically straight. The output surface S is a Lang surface constructed on T. See Fig. 7.6.

Definition 7.5.1. *A **Lang surface** is a surface formed by joining a collection of extrusion disks and rings using the combination and extension operations.*

The boundary polygon for a Lang surface S is, by definition, a doubling polygon for a tree T, which we call its ***boundary tree***. We say that S is ***constructed on*** T. Note that each Lang surface is ***tree projectible***, meaning that its projection onto the xy-plane is a tree T', which is combinatorially equivalent to its boundary tree and geometrically contains it. It should also be noted that different embeddings of the boundary tree give rise to the same intrinsic surface. This entails, in particular, that a continuous motion of the boundary tree in the plane corresponds to a continuous

motion of the Lang surface that maintains the shape of each of its faces. We can then align all of the arcs of the tree along a single line, which "lines up" the boundary edges along a single axis. For this reason, a Lang surface is called ***uniaxial***, a term used by origamists to describe structures of this type.

7.5.2 Properties of zero-curvature Lang surfaces

We are primarily interested in conditions under which the operations described above produce flat Lang surfaces, since these serve as a generalization of flat, polygonal sheets of paper. For the combination operation, it is neccessary and sufficient that both input surfaces be flat. For the extension operation, it is necessary and sufficient that (1) the input Lang surface is flat (as we have seen, all extrusion rings are flat by definition), and (2) the sum of the two angle sums at each vertex along the gluing path is 2π.

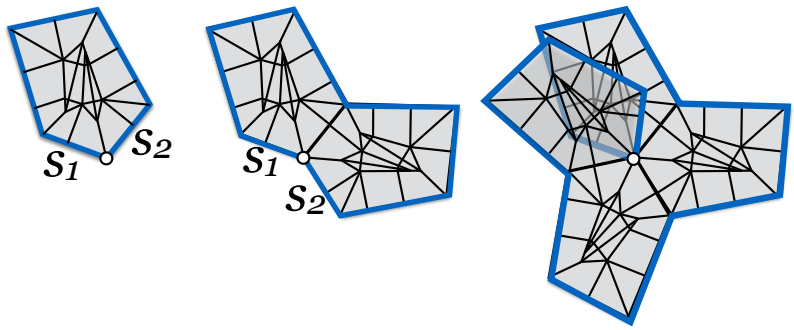


Figure 7.7: Constructing a Lang surface with a boundary vertex v of high curvature using successive combination operations. (a) A single "unit" Lang surface, shown intrinsically. (b) The boundary chains s_1 and s_2 match metrically and combinatorially and we can glue a copy of the surface to itself. (c) Iteratively, we arbitrarily increase the angle sum at v.

Negative boundary curvature. Having removed the restriction on convex boundary polygons, we may now obtain flat Lang surfaces with non-convex boundary. These may even with negative curvature along the boundary. This never occurs as the result of an extension operation, because, as we have seen, the lower boundary of an extru-

sion ring always has non-negative curvature; however, we can perform an arbitrary number of consecutive combination operations, which allows us to make the angle sum at a boundary vertex as large as we want. An example is shown in Fig. 7.7. In this example we took a Lang surface S that flattens out to a convex polygon. The polygon has two sides s_1 and s_2 that are combinatorially and geometrically equivalent, and so we can make arbitrary copies of S and glue them together via the combination operation by gluing s_1 in one copy to s_2 in the first. This arbitrarily increases the curvature at the vertex between s_1 and s_2. We depict these in the flat open state, rather than in the realization as a Lang surface because it is easier to visualize what is going on.

We note two properties of flat Lang surfaces with negative curvature on the boundary. First, as soon as we have introduced negative curvature to the boundary, we can no longer apply extension operations, since to do so would necessarily involve creating an interior vertex of negative curvature, and thus the resulting surface would not be flat. Second, the boundary polygon of this surface is, by construction, a well-formed doubling polygon (in the sense of Sec. 7.3).

Necessity of the geodesic Lang property. The main result of this section can now be described: the boundary polygon P_T of a Lang surface S for a tree T is a geodesic Lang polygon for T. The proof is the same as in Lang's original paper [44], the main difference being that we use geodesic paths rather than straight line segments. We reproduce it here for completeness:

Lemma 7.5.2 ([44])**.** *Let S be a Lang surface for a tree T and P_T be its boundary polygon. Then (T, P_T) is a geodesic Lang polygon on S.*

Proof. Let p denote the geodesic path between any two vertices $\mathbf{u}$ and $\mathbf{v}$ of P_T. In the realization of S, p is a polygonal path in $\mathbf{R}^3$. Recall that the projection of S onto the xy-plane is an embedding of T, thus the projection of p onto the xy-plane contains the path from u to v in T. The projection of p has length less than or equal to the

length of p, which proves that the distance between u and v in T is less than or equal to the distance in S between $\mathbf{u}$ and $\mathbf{v}$. □

7.5.3 The extrusion process as a generalized sweep

We have seen how each extrusion surface (the building blocks of a Lang surface) is generated by tracing a polygon as its edges grow or shrink. Let us take an extrusion surface, say an extrusion ring R, that is used as a building block for a flat Lang surface and "replay" the motion of the extrusion disk across the surface. We do this from the bottom up. We observe that each edge of the extruding polygon, across the face of R it generates, moves in such a way that it remains (intrinsically) parallel to its original position. See Fig. 7.8 (left). We call this the ***extrusion sweep***.

We now define an ***extrusion sweep*** of an entire Lang surface S recursively as follows. In all cases the extrusion sweep starts as the boundary polygon of S. If S is formed by an extension operation on a Lang surface S' and a ring R, then we first perform the extrusion sweep of R, and then recursively continue the extrusion sweep of S'. If S is formed by a combination operation on Lang surfaces S_1 and S_2, then the sweep is defined by first splitting the sweep polygon along the gluing edge between S_1 and S_2, and then continuing the sweep independently in each. In the base case that S is an extrusion surface, we simply perform the extrusion sweep of S and stop. Note that by construction the state of the extrusion sweep at time t is equivalent to the intersection of the $z = t$ plane with S. The edges of S are given by the trace of the vertices of the sweep together with the splitting edges introduced for combination operations. The faces of S are the traces of the edges of the sweep. We illustrate this in Fig. 7.8. In the figure we show a Lang surface that is formed by combining two extrusion disks using a combination operation and then by extending the resulting surface with an extrusion ring. When we replay the extrusion surface "from the bottom up" we first (intrinsi-

cally) perform a parallel sweep of of the ring, then split the sweep into two, and then simultaneously perform parallel sweeps of the the two extrusion disks. We now show:

Lemma 7.5.3. *The extrusion sweep of a Lang surface is a generalized sweep of its boundary polygon and tree.*

Proof. Suppose we perform an extrusion sweep of a Lang surface S constructed on T as defined above. By construction, this sweep is a ***parallel sweep*** (as defined in Sec. 7.4) of the boundary polygon P_T of S with splitting events that occur at discrete events (namely when the sweep hits the base of a part of S formed by the combination operation). To prove the result we need to show that when we attach the appropriate speeds to T making it kinetic and perform the extrusion sweep while shrinking the tree that (1) it maintains the Lang property throughout the sweep and (2) a polygon in the extrusion sweep is split only when the Lang property holds with equality.

To prove claim (1) we first note that the speed assigned to each leaf arc in T in a generalized sweep of (T, P_T) across S is the same speed at which the same leaf arc shrinks during the extrusion process. The claim then follows by the same argument that geodesic Lang property holds on S.

Claim (2) then follows the definition of the extrusion sweep, and the definition of the combination operation. The definition of the combination operation guarantees that the gluing path is straight, and the length of the path is equal to the corresponding distance in the tree. This gluing path is, by definition, what is used to split the extrusion sweep polygon. From this and claim (1) we have that the geodesic Lang property holds with equality for the the pair of vertices on which we split and thus the extrusion sweep is a generalized sweep. □

Having shown that the extrusion sweep of a Lang surface is a generalized sweep, we now argue that every generalized sweep corresponds to the extrusion sweep of some Lang surface.

Lemma 7.5.4. *Let (T, P_T) be a Lang polygon. Then there exists a Lang surface S_T constructed on T such that the extrusion sweep of S_T is the generalized sweep of (T, P_T).*

Proof. To prove this we first associate an *event tree* with the generalized sweep of (T, P_T) (this is, in a sense, the inverse of the construction tree from Sec. 5.2.7). Each event is represented by a node in the tree and the sweep between events is represented by a directed arc in the tree pointing from the later event to the earlier (in other words, arcs represent a parent relationship among events). A splitting event results in multiple incoming edges, one for each of the split polygons in the sweep, whereas a contraction event (that is not also a splitting event) results in one single incoming edge. The root node of the tree is a special starting event denoting the initial boundary polygon, and each leaf is an event at which the one of the sweeping polygons shrinks to a single point as is stopped. Note that in the event tree we only represent splitting events when the polygon and tree are actually split.

We now prove the lemma by induction on the depth of the event tree. Essentially we show that each edge of the tree corresponds to an extrusion surface and each node of the tree corresponds operations on Lang surfaces.

The base case is when the event tree has one edge. This means that only one event occurs in the generalized sweep, namely a contraction of all edges simultaneously to a single point. Let t be the time at which this event occurs. Now, let S_T be the extrusion disk of height t constructed on T. Let f be the face traced by an edge e of the generalized sweep in (T, P_T). It follows from the definitions of the generalized sweep and the extrusion disk that face traced by the corresponding edge in S_T is congruent. It follows from this that S_T is the same intrinsic surface as P_T and that the generalized sweep of (T, P_T) is equal to the extrusion sweep of S_T.

Now assume that the lemma is true for all generalized sweeps with event trees of depth d, we show that it is true for those of depth $d + 1$. Let (T, P_T) be a geodesic

Lang polygon for which the depth of the tree is $d+1$. There are two cases we need to handle, the first is when we split at time $t = 0$ in the generalized sweep of (T, P_T) and the second is when the first event occurs at some time $t > 0$. In the first case, the polygon P_T is split into polygons $P_1, \ldots, P_k$ at time $t = 0$. Each of these correspond to a sub-tree of the event tree. Thus by inductive hypothesis, there are Lang surfaces $S_1, \ldots, S_k$ corresponding to $P_1, \ldots, P_k$. The result then follows by observing that inverse of the splitting process on P_T is the combination operation on $S_1, \ldots, S_k$. In the second case, the root node of the event tree has in-degree one. That edge corresponds to a sweep from P_T to $P_T(t)$. By the same argument as in the base case above, the surface traced by the sweep between P_T and $P_T(t)$ is equivalent to the extrusion ring R constructed on T where the speeds assigned to T for the extrusion process are the same as for the generalized sweep. Then by inductive hypothesis, we have a Lang surface S' constructed on $T(t)$ and observe that applying the combination operation to S' and R gives us a Lang surface S_T constructed on T for which the extrusion sweep is equivalent to the generalized sweep of (T, P_T). □

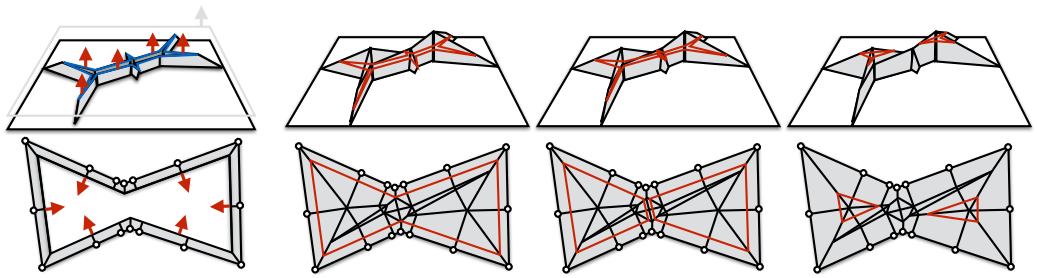


Figure 7.8: The extrusion sweep of a ring (left) and of an entire Lang surface (right).

Summary. We have defined a family of surfaces, called Lang surfaces, that are built on top of a tree. A Lang surface is formed by gluing together extrusion disks and rings using the extension and combination operations. The boundary of a Lang surface is a geodesic Lang polygon. We then put the family of zero-curvature Lang surfaces into correspondence with the generalized sweeps of a Lang polygon by show-

ing that the extrusion process that generates a given Lang surface corresponds to a generalized sweep of its boundary polygon (Lemma 7.5.3), and that a generalized sweep of a Lang polygon always corresponds to the extrusion process for some Lang surface (Lemma 7.5.4). In the next section we describe an algorithm for simulating the sweep. Finally, we show that for any given Lang polygon there is a unique generalized sweep, which puts the Lang polygons into one-to-one correspondence with the zero-curvature Lang surfaces.

7.6 Computing the Geodesic Universal Molecule

In the previous section we defined Lang surfaces, showed that the boundary of a Lang surface is a geodesic Lang polygon, and showed that the extrusion processes is equivalent to a generalized sweep. We now go in the opposite direction. We start with a geodesic Lang polygon (T, P_T) and perform a particular generalized sweep of the polygon maintaining that the sweeping polygon is a geodesic Lang polygon. So far we have not shown that there is a unique generalized sweep for a Lang polygon; instead we have stated that it is possible that at a potential *splitting event*, we may be able to choose whether to actually split or not and in either case maintain the geodesic Lang property. Regardless of the choice we make, we get a generalized sweep. The algorithm described below simulates one particular generalized sweep–namely the one in which we always split at a potential splitting event, even if doing so is not necessary for maintaining the geodesic Lang property. We showed in Lemma 7.4.2 that this sweep must exist. We use it to compute a subdivision of a geodesic Lang polygon into vertices, edges, and faces. We call this subdivision the ***geodesic universal molecule*** of (T, P_T). The edges of the subdivision are given by tracing the vertices of the sweeping polygon (along with the edges introduced at a splitting event).

In this section we describe an algorithm for simulating the sweep and computing the geodesic universal molecule. Then in Sec. 7.7.1 we prove that there exists a Lang

surface with the same boundary Lang polygon and generalized sweep (using the connection between generalized sweeps and extrusion sweeps given in Lemmas 7.5.3 & 7.5.4). Finally, in Sec. 7.7.2 we show that there is *exactly one generalized sweep* for any given Lang polygon (T, P_T). This implies a one-to-one correspondence between Lang surfaces and geodesic universal molecules.

Let us note again that the sweep in the convex case is a generalization of the straight skeleton sweep [4]. For non-convex polygons, the straight skeleton sweep may encounter an event in which a reflex vertex "hits" another edge of the sweep necessitating a split (not to be confused with our splitting events). *We emphasize that such an event cannot occur in our case, because our sweep maintains the invariant that the sweeping polygon is a geodesic Lang polygon, and therefore by Lemma 7.4.1 remains simple. This implies that before such an event occurs a splitting event must precede it.*

The reader should keep in mind that this sweep is performed intrinsically on the surface of P_T; however, in the case that P_T flattens out onto a simple polygon in the plane (convex or non-convex), then we can perform the sweep explicitly in the plane. The algorithm we describe solves the following:

Geodesic Universal Molecule Problem: *Given a geodesic Lang polygon (T, P_T) compute a flat Lang surface constructed on T that is intrinsically equivalent to P_T.*

The algorithm. The *input* is a geodesic Lang polygon (T, P_T) and the *output* is a planar graph G embedded on the surface of P_T such that the subdivision of P_T induced by G is (intrinsically) equivalent to a Lang surface S. We call G the ***geodesic universal molecule*** for (T, P_T). We assume the existence of primitive operations for computing (intrinsic) parallel offset polygons at a given height h and a predicate for determining whether two vertices of a polygon are a visible pair.

The algorithm follows the basic procedure similar to the case of convex, planar Lang polygons. We make the tree T kinetic by attaching a stretching speed $s(ab)$ to

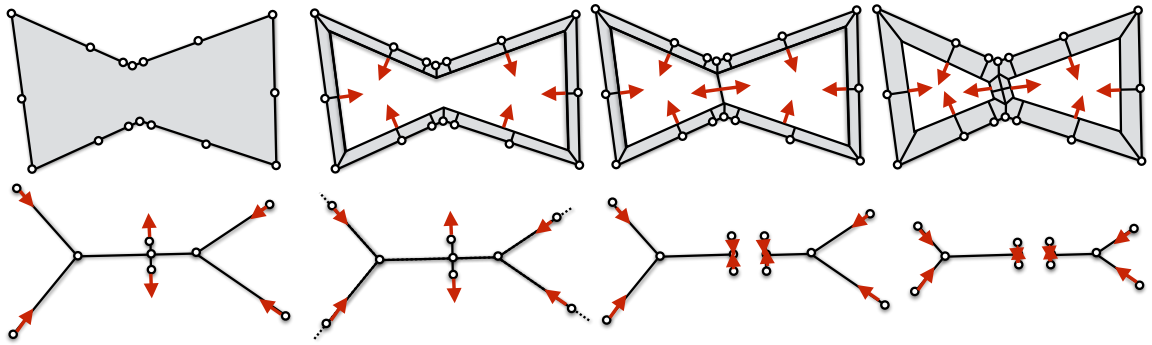


Figure 7.9: A visualization of the sweep of an input geodesic Lang polygon (T, P_T) including a splitting event.

each leaf arc ab. The stretching speed is determined with respect to interior angle of $\mathbf{a}$ in P_T, so that the arc ab maintains the same length in T (as it grows/shrinks) as the length of the corresponding edge in the sweeping polygon. By elementary trigonometry, we have that $s(ab)$ must be $-1/\tan(\theta_{\mathbf{a}})$ where θ_a is the interior angle measure of $\mathbf{a}$ in the sweeping polygon. We refer to the kinetic process in the tree and parallel sweep in the polygon collectively as ***the sweep***.

Recall that for the sweep to be a generalized sweep it maintains the invariant that the kinetic tree and sweeping polygon form a geodesic Lang polygon (Sec. 7.4). Maintaining this invariant requires processing two types of events. A ***contraction event*** occurs when an arc of T and its corresponding edges in P_T shrink to zero length. In this case we remove (or contract) the zero length arcs/edges. A ***splitting event*** occurs when for two non-consecutive corners in P_T, say $\mathbf{a}_i$ and $\mathbf{a}_j$, the geodesic Lang property holds with equality ($d_{P_T}(\mathbf{a}_i, \mathbf{a}_j) = d_T(a_i, a_j)$). As a consequence of Lemma 7.3.1, this pair $(\mathbf{a}_i, \mathbf{a}_j)$ is a visible pair. We then split the tree between a_i and a_j and split the polygon between $\mathbf{a}_i$ and $\mathbf{a}_j$ to obtain a left tree and polygon (T_L, P_L) and right tree and polygon (T_R, P_R) both of which form geodesic Lang polygons with the visible pair now on the boundary. Finally, we continue the sweep independently in each. The output crease pattern is the union of the trace of the vertices throughout the sweep and the splitting edges introduced at a splitting event. See Fig. 7.9 for a

visual overview of the algorithm. The sweep process is simulated by the following recursive procedure:

Listing 7.1: GeodesicUMAlgorithm(T, P_T)

```
function GeodesicUMAlgorithm(T, P_T):
    if (T, P_T) is a base case:
        return handleBaseCase(T, P_T)

    nextEvent := findNextEvent(T, P_T)

    (T', P_T'), R := AdvanceSweepAndTileRing(T, P_T, nextEvent)

    if nextEvent is a contraction event:
        (T', P_T') := Contract(T', P_T')
        G' := GeodesicUMAlgorithm(T', P_T')
    else:
        (T_L, P_L), (T_R, P_R) := Split(T', P_T', nextEvent)
        G_L := GeodesicUMAlgorithm(T_L, P_L)
        G_R := GeodesicUMAlgorithm(T_R, P_R)
        G' := MergeCreasePatterns(G_L, G_R)
    endif

    G := MergeCreasePatternWithRing(G', R)
    return G
```

Simulating the sweep. Each call to the recursive procedure in Listing 7.1 takes as input a Lang polygon (T, P_T) and returns a crease pattern "filling in" P_T. The algorithm first computes the height of the next event by calling the `findNextEvent` subroutine. `AdvanceSweepAndTileRing` advances the sweep to the time of the next event in both the tree and in the polygon to obtain the tree T' and sweep polygon P'_T. This subroutine also produces the tiling R of the annular region between P_T and P'_T given by tracing the vertices of the sweep. If the next event is a contraction event, then the zero-length arcs and edges are contracted in T' and P'_T, and the algorithm is recursively invoked to simulate the sweep in (T', P'_T) and returns a crease pattern G' on the interior of P'_T. Otherwise, the next event is a splitting event for a pair of corners $\mathbf{a}_i$ and $\mathbf{a}_j$. We note that by Lemma 7.3.1, $(\mathbf{a}_i, \mathbf{a}_j)$ must be a visible pair. The algorithm then splits T' and P'_T between a_i and a_j (in the tree) and $\mathbf{a}_i$ and $\mathbf{a}_j$ (in the polygon) to form tree-polygon pairs (T_L, P_L) and (T_R, P_R) (for the left and right sides of the split).

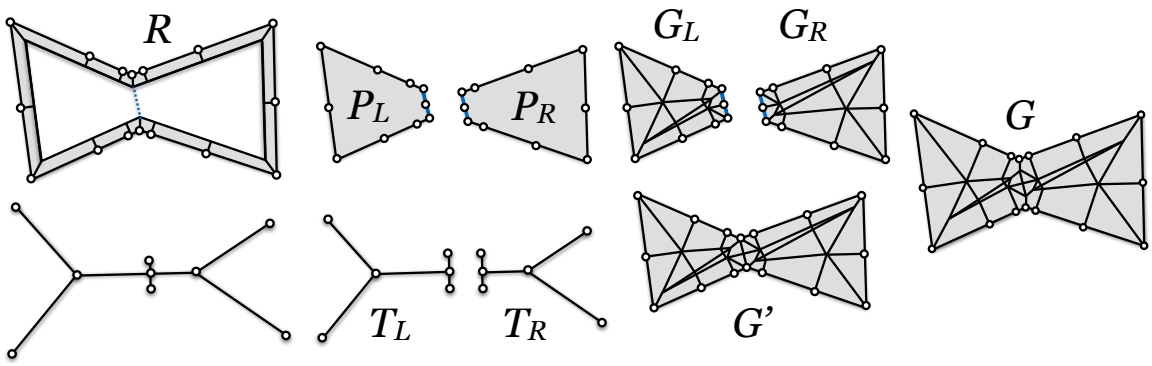


Figure 7.10: An illustration of the recursive procedure in Listing 7.1. We first compute the ring R. Then contract or split (in this case split), fill in each side of the split recursively, and merge this with R to produce an output.

The algorithm is recurisvely invoked on each pair to simulate the sweep in each and obtain crease patterns G_L and G_R for the interiors of P_L and P_R (resp.). These are then merged along the splitting edge between $\mathbf{a}_i$ and $\mathbf{a}_j$ to form a crease pattern G' for the interior of P'_T. Finally, in either case we have a crease pattern G' for the interior of P'_T, and the tiled ring R between P_T and P'_T. We merge R with G' to obtain a crease pattern G on the interior of the input polygon P_T and return the result. See Fig. 7.10.

Computing the events Since corresponding leaf arcs and polygon edges shrink/-grow at the same rate, to find the next ***candidate contraction event*** it suffices to check each leaf arc ab for the smallest value of t for which $d_T(a,b) - 1/\tan(\theta_{\mathbf{a}}/2) = 0$ over all leaf arcs ab.

Finding the next splitting event is a bit trickier. We first note that since we maintain that the sweeping polygon and tree form a geodesic Lang polygon, by Lemma 7.4.1 we never get to a point where a reflex vertex "hits" another edge of the polygon as occurs in the case of the related straight skeletons of a non-convex polygon. By Lemma 7.3.1, the next splitting event must occur for a visible pair; however, as the sweep progresses the set of visible pairs changes, and there is no guarantee that the current visible pairs will be visible at the next splitting event. Let $\bar{P}_T$ denote the open, flat realization of P_T, $\bar{\mathbf{a}}_i$ be the position of $\mathbf{a}_i$ in $\bar{P}_T$, and $\bar{V}_{\mathbf{a}_i}$ denote the velocity of $\bar{\mathbf{a}}_i$ in

the sweep of $\bar{P}_T$. For each pair of corners $(\mathbf{a}_i, \mathbf{a}_j)$ find the smallest value of t satisfying

$$||(\bar{\mathbf{a}}_i + t\bar{V}_{\mathbf{a}_i}) - (\bar{\mathbf{a}}_j + t\bar{V}_{\mathbf{a}_j})|| = d_T(a_i, a_j) - t(1/\tan(\theta_{\mathbf{a}_i}/2) + 1/\tan(\theta_{\mathbf{a}_j}/2)). \quad (7.6.1)$$

The left side of Equation 7.6.1 is the *Euclidean distance* between the two vertices in the flattened sweep at time t and the right side is the corresponding tree distance in the shrinking tree. Solving for t gives the time at which the Euclidean distance becomes equal to the shrinking tree distance. For a visible pair the geodesic distance in P_T and the Euclidean distance in $\bar{P}_T$ are identical. Thus Eq. 7.6.1 is valid for checking the geodesic Lang property for any visible pair. We still do not know which pair will be visible. We check this naively by sorting the times t satisfying Equation 7.6.1 over all pairs of non-consecutive corners in P_T. Then, in increasing order over the times t_i of these candidate events, advance the sweep to each time t_i, and check whether the corresponding pair of corners $(\mathbf{a}_i, \mathbf{a}_j)$ is visible in the sweep polygon at t_i. The first pair we find (smallest value t_i) is the pair for the ***candidate splitting event***. The next event is whichever candidate event (contraction or splitting) occurs at a smaller time t.

Base cases. The base case is when the next event is a contraction of all leaf arcs simultaneously to a single node. (And the degenerate case where all leaf arcs contract simultaneously leaving a path, rather than a single node, in the tree.) This can only occur if all arcs are shrinking, and thus the sweeping polygon is convex. This is the same as in the planar, convex case and occurs if and only if all the angle bisectors intersect at a single point.

Analysis. The recursive algorithm we describe above does not simulate all parts of the sweep simultaneously. When the sweep is split into two it first recursively simulates the sweep on the interior of one side of the split, and then simulates the sweep on the interior of the other side of the sweep. We store the output universal

molecule G, and the tiled ring R as a doubly-connected edge list (DCEL) [5]. Using this structure, computing the ring R, splitting a polygon, and merging crease patterns trivially requires $O(n)$ time. Contracting an edge requires $O(1)$ time. Thus, the main work of each recursive invocation is computing the time of the next event.

We analyze here the naive method. Let n denote the number of nodes in the input tree. To find the next event, we first compute the time at which each edge would contract (assuming no other event occurred first) in constant time per edge. The minimum is the candidate next contraction event. This takes $O(n)$ time. Then, for each pair of non-adjacent vertices, we compute the time at which Eq. 7.6.1 is satisfied to obtain a list E of candidate splitting events. We then sort the list, which takes $O(n^2 \log n)$ time (since there are $O(n^2)$ candidate splitting events). We then look at the first candidate splitting event, compute a representation of the polygon at that event, and check whether the candidate splitting pair of vertices is still visible when the sweep reaches that point. This takes $O(n)$ time using standard techniques. If the candidate splitting pair is visible, then we have found the candidate next splitting event. Otherwise, we check the second candidate event in E, and so on. We continue until we have either found a candidate splitting event for which the pair is visible, or the event time of the candidate splitting event we are considering is greater than the event time of the next candidate contraction event. In the worst case, then, finding the time of the next event requires $O(n^3)$ time, since there are $O(n^2)$ candidate splitting events, and testing the visibility pair in each requires an additional $O(n)$ time. Thus one invocation of the algorithm takes $O(n^3)$ time total and $O(n^2)$ space.

To bound the running time of the algorithm, then, we need to bound the number of events that occur during the sweep. In order to reduce the number of events that occur, it is convenient to compute only the traces of the corner vertices. In other words, we compute the UM-backbone for the geodesic universal molecule as we did in Ch. 6. The traces of the markers can then be computed in a post-processing step.

By the same reasoning as in the convex case, the number of actual contraction and splitting events is $O(n)$ leading to an $O(n^4)$ time algorithm. We note, however, that the naive method described above recomputes almost all of the same candidate splitting events on each recursive call. Using the same techniques as in Ch. 6 this can be improved to $O(n^3 \log n)$ time.

7.7 Proof of main theorem

We now prove the main theorem:

Theorem 7.1.1. *Let P_T be a doubling-polygon for a tree T on a flat, disk-like piecewise-linear surface D. Then a Lang surface S constructed on T and isometric to P_T exists (and is unique) if and only if (T, P_T) is a geodesic Lang polygon.*

Proof Outline. Let (T, P_T) be a geodesic Lang polygon. We show in Lemma 7.7.2 in the next section that the subdivision G returned by Alg. 7.1 is equivalent to a Lang surface S_T constructed on T with boundary polygon P_T. In particular this proves that for any geodesic Lang polygon there exists a Lang surface constructed on T which has P_T as its boundary polygon, namely the Lang surface that is equivalent to the geodesic universal molecule of (T, P_T).

We have already established a correspondence in the other direction via Lemma 7.5.2, which shows that the boundary of each Lang surface is a geodesic Lang polygon.

It remains to establish a one-to-one correspondence, which we do by showing that there exists exactly one generalized sweep for any given Lang polygon, which implies that for a given Lang polygon (T, P_T), there exists a *unique* Lang surface S_T constructed on T that has P_T as its boundary. We prove this in Lemma 7.7.3 in Sec. 7.7.2.

Thus, by the one-to-one correspondence established above, if we are given a geodesic Lang polygon (T, P_T), then there exists a Lang surface S_T constructed on T that is isometric to P_T, and by Lemma 7.5.2, if we start with a Lang surface S_T constructed on T and isometric to P_T, then (T, P_T) is a geodesic Lang polygon. □

7.7.1 Proof of Algorithm Correctness: Geodesic Universal Molecule Crease Patterns are Lang surfaces

We now prove that the algorithm in Listing 7.1 correctly simulates a generalized sweep of the input (T, P_T) and that the resulting subdivision G returned by the algorithm is equivalent to a Lang surface S_T constructed on T with boundary polygon P_T.

Lemma 7.7.1. *Algorithm 7.1 simulates a generalized sweep.*

Proof. By definition the algorithm computes advances a parallel sweep in P_T and grows/shrinks the leaf arcs of T with the same speed assigned to each leaf arc as in the definition of a generalized sweep. Thus what we need to show is that the sweep maintains the geodesic Lang property throughout this sweep (i.e. is a *Lang* sweep). Assume not. Then at some intermediate point in the simulated sweep between two consecutive events processed by the algorithm the Lang property is violated. But the algorithm always processes contraction events it encounters, and any splitting events for visible pairs (since Eq. 7.6.1 gives the time of a splitting event). Therefore, the Lang property was violated for a non-visible pair but not for a visible pair contradicting Lemma 7.3.1. Therefore the sweep simulated by the algorithm is a generalized sweep. □

Lemma 7.7.2. *The subdivision G returned by Alg. 7.1 on a geodesic Lang polygon (T, P_T) is the same subdivision of P_T into vertices, edges, and faces as a Lang surface S_T constructed on T with P_T as its boundary polygon.*

Proof. By Lemma 7.7.1 the algorithm simulates a generalized sweep of (T, P_T). The trace of the vertices of the sweep together with the splitting edges introduced at splitting events induce the subdivision G. By Lemma 7.5.4, this generalized sweep is equivalent to an extrusion sweep of a Lang surface S_T constructed on T with P_T as its boundary polygon. By definition all the edges of S_T are given by the traces of the vertices of the extrusion sweep and the splitting segments introduced at splitting

events. Thus G induces the same subdivision of P_T into vertices, edges, and faces as those given by the construction of S_T. □

7.7.2 Uniqueness

Thus far, we have seen that the vertices, edges, and faces of each flat Lang surface are defined by an extrusion sweep, which, as we saw in Lemma 7.5.3, is a generalized sweep of the surface's boundary polygon and tree. We have also seen that any generalized sweep of a flat geodesic Lang polygon is equivalent to an extrusion sweep of some Lang surface. In Sec. 7.6, we gave an algorithm for producing at least some of the flat Lang surfaces, by simulating one particular generalized sweep of a Lang polygon, namely the one in which we always process splitting events, regardless of whether or not it is necessary to do so to maintain the geodesic Lang property on the sweep. We now show that this is the *only possible* generalized sweep of a flat geodesic Lang polygon. In other words, there is no "choice" to make. If we fail to split the polygon and tree at a potential splitting event, then whatever the resulting sweep is it is not a generalized sweep, since it fails to maintain the geodesic Lang property. Summarizing,

Theorem 7.7.1. *Let (T, P_T) be a flat geodesic Lang polygon. Then there exists a unique generalized sweep of (T, P_T).*

We prove this presently but first note that as a direct consequence we have the following, which is the final step in the proof of the Main Theorem:

Lemma 7.7.3. *Let (T, P_T) be a flat geodesic Lang polygon. Then there exists a unique Lang surface S_T constructed on T that is isometric to P_T.*

Proof. Assume not. Then there are at least two different Lang surfaces with P_T as its boundary, and thus two different extrusion sweeps. But by Lemma 7.5.3, these constitute two different generalized sweeps of (T, P_T), contradicting Theorem 7.7.1. □

The remainder of this section concerns proving Theorem 7.7.1. Our main work is to show that when a generalized sweep encounters a potential splitting event, it *must actually split* in order to maintain the geodesic Lang property. *We note that this proof is significantly more involved than in the convex case.* The proof is based on elementary geometry and vector calculus and requires a careful case analysis of 36 possible cases, only one of which is the convex case. We first outline of the proof of Theorem 7.7.1 and then fill in the details in the theorems and lemmas that follow.

Proof Outline of Theorem 7.7.1. The possibility of the existence of multiple generalized sweeps for the same geodesic Lang polygon (T, P_T) comes from our distinction between potential and actual splitting events. We have thus far allowed that so long as the geodesic Lang property is maintained we do not care if the sweep is *actually split* at each potential splitting event. There are two possibilities we need to consider regarding potential splitting events that may give rise to multiple generalized sweeps for the same geodesic Lang polygon.

First, it may be the case that if multiple potential splitting events occur simultaneously, then actually splitting across one event removes one of the others as a potential splitting event. Theorem 7.7.2 shows that this does not occur. Thus, when we arrive at simultaneous potential splitting events $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{w}, \mathbf{x})$, splitting at one, say $(\mathbf{u}, \mathbf{v})$ leaves the other as a potential splitting event. In other words both $\mathbf{w}$ and $\mathbf{x}$ are in the same split polygon/tree after splitting at $(\mathbf{u}, \mathbf{v})$ which entails that we still have a potential splitting event, since splitting does not effect the distances between $\mathbf{w}$ and $\mathbf{x}$ in the either the polygon or tree.

Second, it may be the case that we can simply ignore some potential splitting events. This would mean that there is a potential splitting event for a pair $(\mathbf{u}, \mathbf{v})$ at some time t such that immediately before the event and immediately after the event the geodesic Lang property is satisfied *if we do not split.* Thus we can choose to either actually split or not to obtain different generalized splitting events. We show

in Theorem 7.7.3 that this is not the case–that whenever we encounter a potential splitting event we *must split* in order to maintain the geodesic Lang property.

Thus, together with Theorems 7.7.2 & 7.7.3, we have that the generalized sweep of a geodesic Lang polygon on a flat surface is unique. □

Event order does not matter. We now show that when simultaneous potential splitting events occur, choosing to split across one of the events does not invalidate any of the others as potential splitting events. This completes the first part of the proof of Theorem 7.7.1 above.

Theorem 7.7.2. *Let* $(\mathbf{u}, \mathbf{v})$ *and* $(\mathbf{w}, \mathbf{x})$ *be simultaneous potential splitting events encountered at some time in a generalized sweep of a Lang polygon. Then after actually splitting for one, say* $(\mathbf{u}, \mathbf{v})$*, the other* $(\mathbf{w}, \mathbf{x})$ *remains a potential splitting event.*

Proof. Let T and P_T denote the tree and sweeping polygon at the time of the event. Let (T_L, P_L) and (T_R, P_R) denote the split polygons obtained by splitting (T, P_T) at $(\mathbf{u}, \mathbf{v})$. Then without loss of generality, either both $\mathbf{w}$ and $\mathbf{x}$ are in P_L or $\mathbf{w}$ is in P_L and $\mathbf{x}$ is in P_R. In the first case, the distances $d_{P_T}(\mathbf{w}, \mathbf{x})$ and $d_T(w, x)$ are not changed by the split and thus $(\mathbf{w}, \mathbf{x})$ remains a potential splitting event. In the second case, we note that the cyclic ordering of $\mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{x}$ along the boundary of P_T is (without loss of generality) $\mathbf{u}, \mathbf{w}, \mathbf{v}, \mathbf{x}$. In other words, the pairs $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{w}, \mathbf{x})$ *cross*, which contradicts Lemma 7.7.4 below. □

The main work of the proof above is Lemma 7.7.4 below, which shows that potential splitting events do not cross. We note that the proof is the same as Lemma 5.5.1 since it relies only on properties of the tree and the triangle inequality. Thus we have:

Lemma 7.7.4. *Potential splitting pairs do not cross in a geodesic Lang polygon.*

The sweep must split at all potential splitting events. We now show that when a generalized sweep encounters a potential splitting event, *it necessarily actually*

splits at the event. Otherwise, we will show, the sweep fails to maintain the geodesic Lang property and thus is not a generalized sweep of a geodesic Lang polygon. This completes the remainder of the proof of Theorem 7.7.1. The proof is significantly more involved than in the convex case. This is because in the convex case, it is fairly straightforward to show that the distances in the plane between vertices of the sweeping polygon always decreases at a rate faster than the corresponding distances in the tree. In the present case, however, distances between points in the sweeping polygon and in the tree may be either increasing or decreasing (depending on the geometry), and it is not necessarily the case that tree distances decrease slower than distances in the sweeping polygon. For this reason, a more careful case analysis is needed. We now prove:

Theorem 7.7.3. *A generalized sweep of a Lang polygon always splits at each potential splitting event.*

Proof. Suppose that we reach a potential splitting event $(\mathbf{a}_i, \mathbf{a}_j)$. We prove the theorem by comparing the rates at which the distances are changing in the sweeping polygon and tree. Let $d_S(t)$ denote the distance at time t between $\mathbf{a}_i$ and $\mathbf{a}_j$ in the sweep and $d_T(t)$ denote the distance between the corresponding leaf nodes a_i and a_j in the kinetic tree. Let $d(t)$ denote the difference between them, i.e. $d(t) = d_S(t) - d_T(t)$. For simplicity, we will shift the event times so that the event occurs at time $t = 0$, and thus $d_S(0) = d_T(0)$, or equivalently $d(0) = 0$. Let $\Delta t > 0$ be a small value near 0. Just before the event (i.e. at time $-\Delta t$), the geodesic Lang property holds, and so $d_S(-\Delta t) > d_T(-\Delta t)$, and thus $d(-\Delta t) > 0$. We want to show that just after the event the geodesic Lang property is violated, i.e. $d(\Delta t) < 0$ for all small enough Δt. To do this, we need to show that $d(t)$ is not at a local minimum at $t = 0$. It suffices to show that it is not at a critical point, meaning that its derivative $d'(0) = d'_S(0) - d'_T(0) \neq 0$, or equivalently $d'_S(0) \neq d'_T(0)$. We prove this in Lemma 7.7.5 below.

Now assume that a generalized sweep encounters a potential splitting event but does not split. By Theorem 7.7.2, we have that the potential event cannot be removed by actually splitting for some other simultaneously occurring potential splitting event. But then, by the discussion above we have that immediately after the event, the geodesic Lang property is violated, contradicting that our sweep is a generalized sweep. □

Lemma 7.7.5. *The instantaneous rate of change in the geodesic distance between two non-consecutive visible corners* $\mathbf{a}_i$ *and* $\mathbf{a}_j$ *in the parallel sweep is not equal to the instantaneous rate of change in the corresponding distance in the kinetic tree.*

Proof Set-up and Outline. Here we only outline the proof of Lemma 7.7.5 and give the geometric set-up. The details of the proof are then organized into the lemmas and theorems below. We then give the full proof at the end this section starting from the paragraph titled "Proof of Lemma 7.7.5".

Initial set-up. As in Theorem 7.7.3, we denote the distance function between $\mathbf{a}_i$ and $\mathbf{a}_j$ in the sweep by $d_S(t)$ and the distance function in the tree between a_i and a_j by $d_T(t)$. To prove the theorem, we show that the instantaneous rate of change in the geodesic distance, $d'_S(0)$ is not equal to the instantaneous rate of change, $d'_T(0)$ in the tree. In principle our argument holds for all values of t so long as $(\mathbf{a}_i, \mathbf{a}_j)$ is a visible pair.

Overview. Instead of deriving a closed form for $d'_S(t)$ and $d'_T(t)$, we show in Lemmas 7.7.6 & 7.7.8 how to determine the value of $d'_S(0)$ and $d'_T(0)$ at $t = 0$ geometrically. We show that the values of $d'_S(0)$ and $d'_T(0)$ are determined by two vectors defined at each vertex, which we label V_i and W_i at $\mathbf{a}_i$ and V_j and W_j at $\mathbf{a}_j$. Then, by analyzing the relative magnitudes of V_i and W_i and the relative magnitudes of V_j and W_j we prove that $d'_S(0) \neq d'_T(0)$. To do this we first initiate a study of the relative magni-

tudes of V_i and W_i at $\mathbf{a}_i$, from which we derive 6 distinct cases, labelled A–F. These depend on whether $\mathbf{a}_i$ is convex or not in the polygon and the angle made between the edges incident to $\mathbf{a}_i$ and the visibility segment $\mathbf{a}_i\mathbf{a}_j$. We then complete the proof of the lemma by showing in all 36 possible cases (where both vertex $\mathbf{a}_i$ and vertex $\mathbf{a}_j$ may be any of the cases A–F), $d'_S(0) \neq d'_T(0)$.

Notation. In the remainder of this section we use upper cased non-bold type with subscripted i or j to denote vectors, such as U_i or U_j, defined at $\mathbf{a}_i$ and $\mathbf{a}_j$ resp. We denote the magnitude of a vector U_i by its lower-case $u_i = ||U_i||$.

Flattened ϵ-patch. In order to simplify the discussion that follows we "flatten out" the polygon and sweep in the plane. This flat realization, as we have seen, may have self-intersections; however, if we restrict ourselves to a small enough patch, say all points within some small ϵ distance of the visibility segment between $\mathbf{a}_i$ and $\mathbf{a}_j$, then the patch is realized in the plane as a small planar region without self intersections. See Fig. 7.11. In the remainder we use this "local" view of the flat realization in the plane, which allows us to use elementary plane geometry to analyze the geometry near the visibility segment.

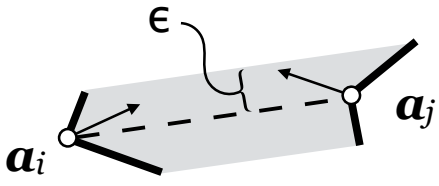


Figure 7.11: An illustration of the flattening out of a small ϵ band around the visibility segment between $\mathbf{a}_i$ and $\mathbf{a}_j$. The dashed line denotes the visibility segment, the dotted line denotes the sweep, and the two arrows denote the motion vectors of $\mathbf{a}_i$ and $\mathbf{a}_j$ in the unit-speed parallel sweep.

Deriving the vectors U_i, V_i and W_i. Let U_i denote the instantaneous velocity vector of $\mathbf{a}_i$. By definition, U_i points along the interior angle bisector at $\mathbf{a}_i$. We now use U_i to define two vectors, V_i and W_i at $\mathbf{a}_i$. Project U_i onto the visibility

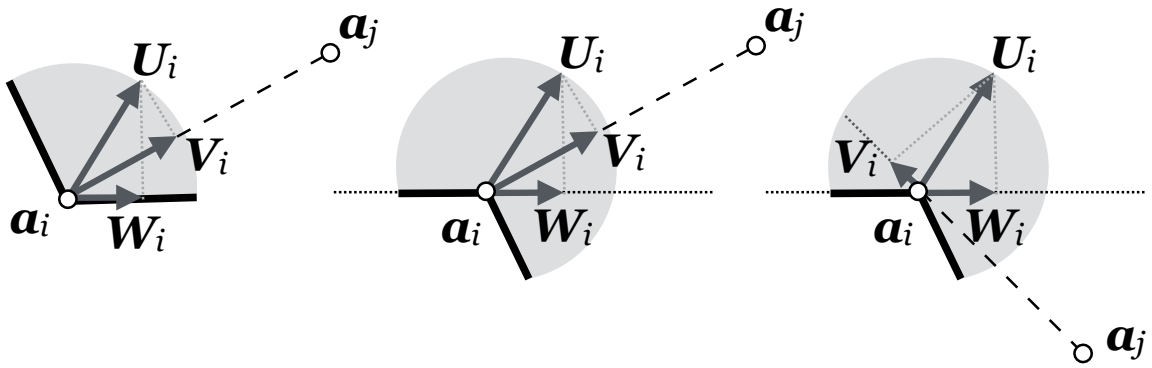


Figure 7.12: An illustration of the vectors defined in the proof of Lemma 7.7.5 for the vertex $\mathbf{a}_i$ in three different situations. The left shows the convex case. The middle shows a reflex case in which $\mathbf{a}_i$ is moving towards $\mathbf{a}_j$. The right shows a reflex case in which $\mathbf{a}_i$ is moving away from $\mathbf{a}_j$. U_i denotes the motion vector of $\mathbf{a}_i$ along the interior angle bisector in the sweep. V_i is the projection of U_i onto the line between $\mathbf{a}_i$ and $\mathbf{a}_j$, and W_i is the projection of U_i onto the line supporting one of its edges.

segment $\mathbf{a}_i\mathbf{a}_j$. Let L denote the supporting line through one of the edges incident to $\mathbf{a}_i$. Project U_i onto L to obtain W_i. Note that because U_i points along an angle bisector, and because we are really only interested in the magnitude w_i and not the direction, it does not matter which edge incident to $\mathbf{a}_i$ is chosen to construct L (the magnitude w_i is the same regardless of the choice). Examples are shown in Fig. 7.12. We define vectors U_j, V_j, and W_j at $\mathbf{a}_j$ similarly.

What remains. In Lemma 7.7.6 we show how $d'_S(0)$ relates to the magnitudes v_i and v_j. In Lemma 7.7.8 we show how $d'_T(0)$ relates to the magnitudes of w_i and w_j. We end with the proof of Lemma 7.7.5. We now summarize a basic property from elementary vector calculus:

Lemma 7.7.6. *The instantaneous rate of change in the distance between $\mathbf{a}_i$ and $\mathbf{a}_j$ in the polygon is given by*

$$d'_S(0) = \pm v_i \pm v_j \tag{7.7.7}$$

where the sign in front of v_i (resp. v_j) is '+' if V_i points towards $\mathbf{a}_j$ (resp. V_j points towards $\mathbf{a}_i$), otherwise the sign is '-'.

We now derive $d'_T(0)$:

Lemma 7.7.8. *The instantaneous rate of change in the distance between a_i and a_j in the tree is given by*

$$d'_T(0) = \pm w_i \pm w_j \tag{7.7.9}$$

where the sign in front of w_i (resp. w_i) is '-' if vertexa_i (resp. $\mathbf{a}_j$) is convex in the polygon, '+' otherwise.

Proof. The vector W_i is the orthogonal component of $\mathbf{a}_i$'s instantaneous motion towards the other endpoint of one of the edges incident to $\mathbf{a}_i$. The proof then follows from the fact that we defined the speeds at the leaf arcs so as to maintain the length in the tree between edges in the sweeping polygon and their corresponding leaf arcs. □

Characterizing the relative magnitudes of v_i and w_i. We now characterize the relative magnitudes of V_i and W_i, and the signs in front of each in Eqs. 7.7.7 & 7.7.9. There are six possible cases, which we label A–F, which depend on whether $\mathbf{a}_i$ is convex or not, and the angle made by the visibility segment $\mathbf{a}_i\mathbf{a}_j$ with the edges incident to $\mathbf{a}_i$. The construction of cases A–F, detailed shortly, is illustrated in Fig. 7.13 and the relative magnitudes of v_i and w_i in each case is summarized in Table 7.1 below. Those for $\mathbf{a}_j$ are similar. Using the table together with Eqs. 7.7.7 & 7.7.9 allows us to determine the values of $d'_S(0)$ and $d'_T(0)$ based on which cases A–F are $\mathbf{a}_i$ and $\mathbf{a}_j$. For instance, if $\mathbf{a}_i$ is case B and $\mathbf{a}_j$ is case D, then using the table and the two equations we see that $d'_S(0) = v_i - v_j$ and $d'_T(0) = w_i + w_j$.

	$v_i ? w_i$	sign(v_i) in $d'_S(0)$	sign(w_i) in $d'_T(0)$
A	$v_i > w_i$	-	-
B	$v_i < w_i$	+	+
C	$v_i < w_i$	$v_i = 0$	+
D	$v_i < w_i$	-	+
E	$v_i = w_i$	-	+
F	$v_i > w_i$	-	+

Table 7.1: A summary of the relationship between the magnitudes v_i and w_i of the vectors V_i and W_i from Fig. 7.12 organized by cases A–F (A is the convex case, cases B–F are illustrated in Fig. 7.13) and the sign in front of each in $d'_S(0)$ and $d'_T(0)$ for all possible cases of the vertex $\mathbf{a}_i$.

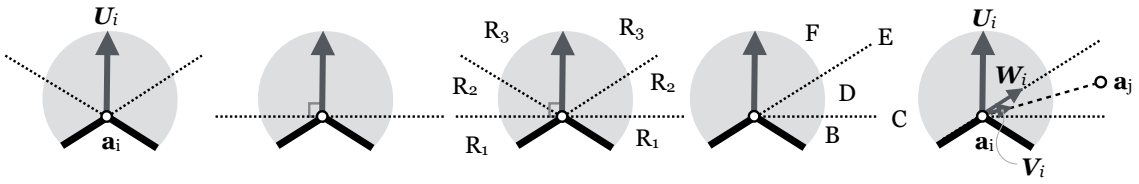


Figure 7.13: Constructing the five non-convex cases B–F used to define the cases in the case analysis in the proof of Lemma 7.7.5. *From left to right, First:* Extend the lines supporting the two edges incident to a vertex (dashed). *Second:* The line (dashed) orthogonal to the interior angle bisector (gray vector). *Third:* The wedge regions R_1 through R_3 defined by the lines extending the edges and the line orthogonal to the interior angle bisector (the wedges to the left of the bisector are symmetric to those on the right). *Fourth:* The labelling of the cases to the right of the interior angle bisector (the left is symmetric). *Fifth:* An example where the position of a_j follows in region R_2, making a_i case D. In this case the length of W_i is greater than the length of V_i.

Constructing cases A–F. Case A is when $\mathbf{a}_i$ is convex, cases B–F are when $\mathbf{a}_j$ is non-convex. To construct cases B–F, first extend lines through the edges incident to $\mathbf{a}_i$. Next construct the line through $\mathbf{a}_i$ perpendicular to the interior angle bisector of $\mathbf{a}_i$. See the first two parts of Fig. 7.13. These three lines together with the interior angle bisector divide the wedge around $\mathbf{a}_i$ into six wedge "slices". Those on the left side of the angle bisector are symmetric to those to the right, so we label them R_1, R_2, and R_3 symmetrically and in the remainder concentrate on the right side–see the center of Fig. 7.13. Case B is when $\mathbf{a}_j$ falls in the wedge R_1. Case C is when it falls on the line between R_1 and R_2. Case C is when it falls in R_2. Case E is when it falls on the line between R_2 and R_3. Case F is when it falls in R_3. We now have:

Lemma 7.7.10. *Table 7.1 summarizes the relationship between v_i and w_i and the signs in front of each in Eqs. 7.7.7 & 7.7.9.*

Proof. The lines extended in the construction of regions R_1, R_2, and R_3 are precisely those where relative magnitudes and the signs in Eq. 7.7.7 change. Extending the lines through the edges in the construction of R_1 through R_2 divides the plane into four regions (see the left-most illustration in Fig. 7.13). One is outside the the polygon.

The next two are incident to the two edges incident to $\mathbf{a}_i$. In these, by elementary geometry it follows that $v_i < w_i$. In the remaining, $v_i > w_i$, and if $\mathbf{a}_j$ lies on the line through one of the edges then $v_i = w_i$. The line through $\mathbf{a}_i$ perpendicular to the interior angle bisector is the dividing line such that if $\mathbf{a}_j$ is below it (i.e. in R_1), then $\mathbf{a}_i$ is moving away from $\mathbf{a}_j$ (and hence v_i has a '+' in Eq. 7.7.7). If $\mathbf{a}_j$ is on the line, then $\mathbf{a}_i$ is moving perpendicularly relative to $\mathbf{a}_i$ and $\mathbf{a}_j$ and so $v_i = 0$. Otherwise $\mathbf{a}_j$ is moving towards it. The table simply summarizes these facts. □

Proof of Lemma 7.7.5. We now complete the proof of Lemma 7.7.5 by showing that for all possible cases A–F of $\mathbf{a}_i$ and all possible cases A–F of $\mathbf{a}_j$, the instantaneous rate of change in the geodesic distance $d'_S(0)$ is not equal to the instantaneous rate of change in the tree.

Proof. We have 36 cases to consider, depending on which cases A–F are the two vertices $\mathbf{a}_i$ and $\mathbf{a}_j$. Each case is labelled with the two case letters for $\mathbf{a}_i$ and $\mathbf{a}_j$. For instance, if vertex $\mathbf{a}_i$ is B and $\mathbf{a}_j$ is D, then we label it BD. Note that symmetric cases use the same proof, so we only list cases in lexicographical order (BD and DB are the same so we use BD). In each case we start by using Table 7.1 to derive $d'_S(0)$. We then use the relationship between v_i and w_i and the relationship between v_j and w_j to show that $d'_S(0) \neq d'_T(0)$ (which is found by plugging in the appropriate values from Table 7.1 into Eq. 7.7.9).

Below we prove each case on a single line, however, to give the reader a full sense of the line of proof we do one expanded case here, the case where $\mathbf{a}_i$ is B and $\mathbf{a}_j$ is E (i.e. case BE). Looking up B for $\mathbf{a}_i$ and E $\mathbf{a}_j$ in Table 7.1 and pluggin the appropriate values into Eqs 7.7.7 & 7.7.9 we have that $d'_S(0) = v_i - v_j$ and $d'_T(0) = w_i + w_j$. We now start with $d'_S(0)$ and show that it is not equal to $d'_T(0)$. $d'_S(0) = v_i - v_j < w_i - v_j$ since by Table 7.1 $v_i < w_i$ for case B. $w_i - v_j < w_i + w_j$ since all magnitudes are positive, and thus subtracting v_i from w_i is less than adding any positive value to w_i.

But $w_i + w_j = d'_T(0)$, and thus $d'_S(0) < d'_T(0)$. We now show the full case by case analysis. All inferences are either derived from Table 7.1 as above, or follow from the fact that all magnitudes are positive.

- AA: $d'_S(0) = -v_i - v_j < -w_i - v_j < -w_i - w_j = d'_T(0)$.
- AB, AC: $d'_S(0) = -v_i + v_j < -w_i + v_j < -w_i + w_j = d'_T(0)$.
- AD, AE, and AF: $d'_S(0) = -v_i - v_j < -w_i - v_j < -w_i + w_j = d'_T(0)$.
- BB, BC: $d'_S(0) = v_i + v_j < w_i + v_j < w_i + w_j = d'_T(0)$.
- BD: $d'_S(0) = v_i - v_j < v_i + w_j < w_i + w_j = d'_T(0)$.
- BE, BF: $d'_S(0) = v_i - v_j < w_i - v_j < w_i + w_j = d'_T(0)$.
- CC: $d'_S(0) = 0$. $d'_T(0) = w_i + w_j > 0$.
- CD, CE, CF: $d'_S(0) = -v_j < w_j < w_i + w_j = d'_T(0)$.
- DD, DE, DF, EE, EF, FF: $d'_S(0) = -v_i - v_j < 0 < w_i + w_j = d'_T(0)$.

The remaining cases are symmetric. In all cases we have shown that $d'_S(0) \neq d'_T(0)$, which proves that $d(t) = d_S(t) - d_T(t)$ is not at a local minimum at $t = 0$ and thus to proceed in the sweep constitutes a violation of the geodesic Lang property. □

Summary. We have shown that the instantaneous rate of change in the geodesic distance between $\mathbf{a}_i$ and $\mathbf{a}_j$ is not equal to the instantaneous rate of change in the tree distance between a_i and a_j. In particular, this shows that the derivative $d'(0) = d'_S(0) - d'_T(0)$ does not vanish, meaning that $d(0)$ is not a critical point. Since $d(0)$ is not a critical point, then to continue the sweep past a splitting event *without splitting* results in a violation of the geodesic Lang property. This completes the proofs of Theorems 7.7.3 & 7.7.1. Thus for any given flat geodesic Lang polygon, there exists a unique generalized sweep fro the polygon. This sweep is precisely the

sweep simulated by the geodesic universal molecule algorithm. Furthermore, together with Lemmas 7.7.1 and 7.5.4 this entails that there is a unique geodesic Lang surface constructed on T having P_T as its boundary and the subdivision of P_T into vertices, edges, and faces given by S_T is the geodesic universal molecule of P_T. This in turn is the final ingredient in the proof of the main theorem, Theorem 7.1.1.

7.8 Conclusion

In this chapter we generalized the universal molecule algorithm to geodesic Lang polygons which form the boundary of piecewise-linear disk-like surfaces in $\mathbf{R}^3$ with zero-curvature. Restricted to simple, non-convex polygons in the plane our algorithm extends the TreeMaker algorithm to cases where before it could not produce an output. A further open problem is an extension of the algorithm to surfaces of non-zero curvature. Our Lang surfaces are more general and can be used to construct surfaces with non-zero curvature. Can the algorithm be extended to compute these from the boundary polygon? If we are given a geodesic Lang polygon drawn on a surface with singular points of non-zero curvature, can we always compute a geodesic universal molecule on the polygon that is equivalent to some Lang surface?

CHAPTER 8

RIGIDITY OF UNIVERSAL MOLECULES

In the previous chapters we have developed algorithms for the straight skeleton (Ch. 4 and the universal molecule (Chs. 5 & 6) and generalized the universal molecule to non-convex polygons (Ch. 7). Our motivation for studying these structures comes primarily from computational origami. In addition to the algorithmic questions we have investigated thus far, a further important question in computational origami is the rigid folding question. Given two realizations of the same origami crease pattern, for instance the universal molecule in its flat planar state and in its folded Lang surface state, does there exist a motion between the two that preserves the surface without cutting it while maintaining the geometry on each face? Possible answers to this question are: there never exists such a motion, there always exists such a motion, or there sometimes exists such a motion, if certain conditions are met.

Prior to this work it was known that in the special case that the universal molecule only encounters contraction events (i.e. it is equivalent to the straight skeleton with added perpendiculars), then there *always* exists a motion between the initial open, flat state and the "folded" Lang surface state [27]. Such cases, however, are extremely rare. In this chapter we show that for a larger class of polygons the initial open, flat state is completely rigid–there does not exist a motion between it and any other non-trivial realization; and the Lang surface state is ***stable***, in the sense that all of the motions it gives rise to correspond motions of the underlying tree, but it can never "open up" in the sense that it always projects onto the tree.

The work in this chapter has previously appeared in [17].

8.1 Introduction

Paneled origami. In this chapter we work with the universal molecule as a *flat-faced origami*[1]. The creased "paper" behaves like a mechanical panel-and-hinge structure, and for this reason we refer to it as ***paneled origami***. The central question we are concerned with for a paneled origami is what motions exist that allow panels to rotate around their edges while maintaining their shape.

Deployable structures. Paneled origami models one of origamis main application areas–so called deployable structures. Deployable structures are structures built for real world uses that fold up for various purposes. For instance, one problem is to design an array of solar panels that are connected together along hinges that can be folded up into a small container. This allows the solar array to be more easily be packed into a space ship and launched into orbit. Once in orbit it can be unfolded into its deployed state. In this (and related) applications, each panel is inflexible (rigid)–it always maintains its shape as a flat polygon. Motion is allowed only at the hinges connecting panels. This is in opposition to paper origami, since paper can not only fold along the "hinges", or creases, but can bend, warp, and stretch throughout a folding process. Understanding the rigid foldability of an origami design tool is important to understanding the possible scope of its applications.

Our goal. Our goal in this chapter is to address a major question in the foldability of the universal molecule: when is the universal molecule, when viewed as such a paneled origami, "foldable"? Neither an efficient algorithm nor a good characterization are known for this decision problem, with the exception of single-vertex origami [50, 49] and some disparate cases, including the "extreme bases" of [27] in which the universal molecule is identical to the straight skeleton.

---

[1]Sometimes called *rigid origami* in the literature. We avoid this terminology because of its potential for ambiguity in the context of this paper.

Our results. We show that only certain special crease patterns *have a chance to be foldable* and identify a combinatorial pattern of a universal molecule (captured in an associated outerplanar graph) that forces it to be rigid in the open state and stable (not *unfoldable*) in the folded Lang surface state. This unexpected behavior of the algorithm puts in perspective some of the most relevant properties of the computed output, and opens the way to design methods that *may* overcome these limitations. Our proof technique, called *rigidity transport*, is algorithmic in nature and, to the best of our knowledge, new. As is the case with similar questions in combinatorial rigidity theory, a *complete characterization* of rigid, resp. stable patterns appears to be substantially more difficult; we leave it as an open question.

Non-degeneracy. In this chapter we deal only with universal molecules that are not degenerate. By ***non-degenerate***, we mean a universal molecule which has no contraction events (other than the base case contraction events) and in which no events occur simultaneously during the parallel sweep process (see Ch. 5). Such universal molecules have nice properties which we detail in the next section. Further, any small perturbation of the coordinates of a degenerate polygon results in a non-degenerate one.

8.2 Preliminaries

Outerplanar graphs. An outer planar graph is a planar graph which can be drawn in the plane such that all vertices are incident to the outer face and no two edges cross. An outer planar graph is made up of a cycle and edges between vertices of the cycle which do not cross. Label the vertices around the cycle in counter-clockwise order by 1, ..., n, then two edges (a, b) and (c, d) cross if c is (cyclically) between a and b and d is (cyclically) between b and a (or vice versa). Such a graph has a canonical face set which is given by any embedding of the graph in which all vertices are incident to the outer face. We will refer to this simply as *the face set* for the outer

planar graph (without referring to any specific embedding). We call a face which is not incident to the outer face in such an embedding *internal*.

Splitting graph. We now associate a particular outer planar graph with a non-degenerate universal molecule, which we call the ***splitting graph***. The initial graph is the polygon itself. Each time a splitting event occurs between two vertices in the sweep we add an edge between the corresponding nodes in the splitting graph. See Fig. 8.2. The splitting graph is outerplanar because splitting pairs do not cross at any point in the sweep (Lemma 5.5.1).

Configuration space, flexibility, and rigidity. Given a realization of an origami pattern, new "trivial" realizations may be obtained by applying euclidean translations and rotations to the entire realization, but the overall "shape" of the realization remains unchanged. Factoring out translations and rotations gives us the *configuration* or *state* of the origami. The space of all possible states is the *configuration space*, which is, more formally, the space of all realizations modulo euclidean translations and rotations. Paths in the configuration space correspond to continuous motions of the origami pattern which keep the faces as rigid panels which rotate around the edges acting as hinges. A path in the configuration space is called a *flex*. An origami pattern in a given state is *flexible* if there exists a flex of the pattern, otherwise it is *rigid*. Rigid states are isolated points in the configuration space–given any other state, no path exists between that state and any rigid state. We use configuration states to define the following concepts which will be used throughout the paper.

Flat and open-flat states. If all faces of a state of the origami are coplanar, then we say that the origami is in a *flat* state. If an origami is flat the dihedral angle of each internal edge of the origami pattern is either 0 or π. The converse is also true. If further, the dihedral angle at each internal edge is π, we say that the base is in the *open, flat* state.

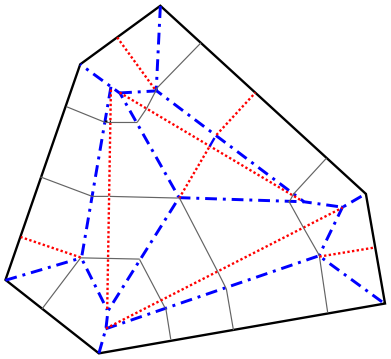


Figure 8.1: Rigid crease pattern. The mountain (blue dot-dash)/valley (red dot) assignment indicates the pattern of the flat-folded Lang base configuration.

8.3 Overview of the Main Results

We have seen that Lang's algorithm produces origami patterns that *have a folding* as Lang surfaces (Ch. 5). We first show that many of these patterns in the open, flat state are in fact rigid: they are isolated points in the origami's configuration space, and therefore do not fold to anything else. The "folded" Lang surface lies in a different component of the configuration space, and, due to its intrinsic tree-like structure, it is obviously flexible. However, we show that it too is isolated, but in a different way, which we call *stable*. Such distinguishing properties, although possibly experienced "intuitively" by origamists, have been neither previously identified nor proven in the literature.

Our main result in this chapter is:

Theorem 8.3.1. (Rigid Universal Molecules) *There exist universal molecules which are rigid in the open-flat state.*

The existence of rigid universal molecules is based on the universal molecule in Fig. 8.1. The coordinates are *generic*, i.e. the pattern does not change under small perturbations. The proof, given in the next section, is also indicative of the fact that its rigidity is not a simple artifact of some rare occurrence or numerical imprecision.

We generalize this example in two ways, first by turning it into a sufficient criterion for detecting the rigidity of the crease pattern, then by extending it to the

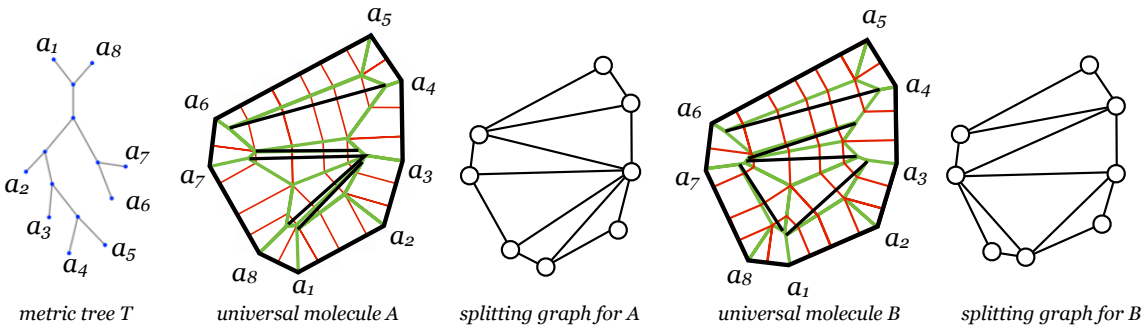


Figure 8.2: A metric tree (left) with two compatible Lang polygons, each one with its universal molecule pattern and associated splitting graph. In the first case, all the faces of the splitting graph are exposed. In the second case, one triangular face is isolated from the polygon boundary. Tiling colors indicate edge types (splitting edges (black), bisectors (green), perpendiculars(red)).

corresponding Lang surface state. This generalization leads to a family of universal molecules distinguished by the existence of a special degree-6 vertex which we call an *isolated peak* (defined in Sec. 8.5). Informally, an isolated peak is a degree-6 vertex of the crease pattern which is "isolated" from the boundary in the manner of the basic example from Fig. 8.1: none of the creases emanating from the vertex reach the boundary of the crease pattern.

Theorem 8.3.2. (Universal Molecules with isolated peaks are rigid) *If a universal molecule has an isolated peak, then it is rigid in the open-flat state.*

Lang surfaces, however, are always flexible in their folded state, inheriting the same degrees of freedom as the trees they are constructed on. A foldable state should be reached through a continuous deformation path from the initial open, flat state. But by Theorem 8.3.2, we know that the universal molecules with isolated peaks lead to Lang bases that cannot be reached from the open state. The question then is can they be reached from some other interesting intermediate configuration? We prove that this is not the case.

Theorem 8.3.3. (Stability of Lang Lang Bases with isolated peaks) *All rigid folding motions of a Lang surface correspond to motions of the underlying tree. In*

other words, the Lang surface cannot be "unfolded" all of its faces remain perpendicular to a common plane throughout any rigid folding motion.

8.4 Rigid Universal Molecules

In this section we prove Theorem 8.3.1 by showing that the universal molecule pattern from Fig. 8.1 is rigid. The main challenge is to prove rigidity in the absence of infinitesimal rigidity. Indeed, infinitesimal rigidity would have implied rigidity, but this is not the case here: an infinitesimal motion, with vertex velocities perpendicular to the plane of the "paper", always exists. For the proof, we introduce a different technique, called *rigidity transport*. It is algorithmic, and can be applied on any graph as long as it has vertices with 4 "unvisited" edges that act as "transmitters" (cf. definition given below) and which are reachable from a starting point via "transport" edges.

Single-vertex origami with 4 creases. The faces surrounding a vertex incident to 4 edges (creases), isolated from the rest of the origami pattern, is called a 4-*edged single-vertex origami*. A generic realization, is flexible, with a one-dimensional configuration space, meaning that when one of the dihedral angles is changed continuously, all the other dihedrals are determined: the origami has a one-dimensional *flex*. The flat open configuration is however not generic, it is singular, and thus may allow flexes to proceed along different branches of the one-dimensional configuration space. We rely on the tabulation of all the types of configuration spaces for planar 4-gons, which can be found for instance in [34], and on the relationship between the Euclidean, spherical and single-vertex origamis, as discussed at large in [50, 49]. We start by identifying in Fig. 8.3 the types of single-vertex origamis with 4-creases (called, for simplicity,"4-edged gadgets") that appear in a Universal Molecule crease pattern.

Rigidity transport. Assume that a dihedral (input) edge of one such 4-edge gadget is kept, rigidly, in the flat open position (of 180^o) or flexed by a small angle: can

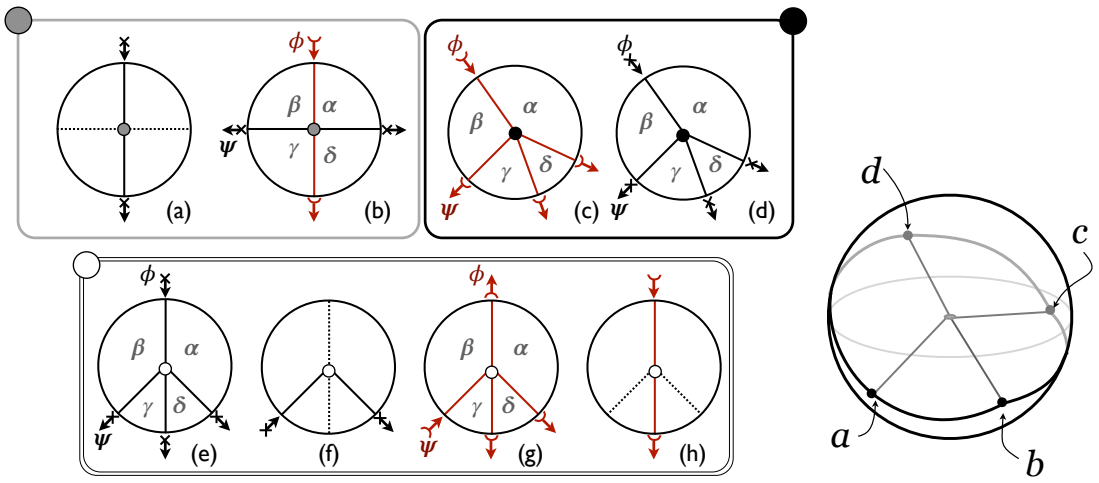


Figure 8.3: *Left:* Rigidity transport of flat 4-edge single-vertex origamis appearing in a universal molecule. Arrows represent "input" and "output" edges. A cross on an arrow's tail indicates the edge remains open-flat, while a small crescent indicates a flex. An edge for which the behavior is not forced by the input edge is dotted. *Right:* The four bar mechanism *abcd* formed by a 4-edge single-vertex origami.

the behavior of the other (output) edges be predicted? For 4-edge gadgets created by the universal molecule algorithm and needed in the main proof of this section, Fig. 8.3 tabulates all the possibilities. The patterns are grouped into three categories: black, white, and gray. In each category opposite angles are supplementary (i.e. $\alpha + \gamma = \beta + \delta = 180^o$). In Fig. 8.3 the category of each gadget is indicated by the shading of the center vertex. An arrow pointing towards the center indicates the "input" to the gadget. For easy reading we denote the "signals" by a marker on the tail of the arrow: a cross indicates that the dihedral edge is kept at 180^o, and a small crescent indicates a slight perturbation (flex). An arrow pointing outwards indicates a forced behavior on another edge. A dotted edge signifies that its behavior is not determined by the input, and that it can either stay flat of have a small flex.

In **white**, one pair of opposite edges are aligned and the other two make equal angles (different from 90^o) with them. In **gray**, both pairs of opposite edges are aligned, and one is perpendicular to the other (i.e. all face angles are 90^o). In **black**, opposite angles are supplementary. We analyze these patterns with respect to an "input" crease, i.e. the mechanical action of keeping the dihedral angle as it is or perturbing

it slightly. In each case if an edge is flexed resp. remains rigid, then its opposite edge is flexed resp. remains rigid; in addition, in **black**, all edges either flex or remain rigid collectively (Fig. 8.3, c, d); in **white** if an aligned edge remains rigid (Fig. 8.3,e) or an unaligned edge flexes (Fig. 8.3,g), then all edges remain rigid or flex (resp.); and in **gray** if an edge flexes, then the opposite pair of edges remain rigid (Fig. 8.3,b).

These categories will be used as follows: (a) the black gadgets will appear at the endpoints of a splitting edge, (b) the white gadgets to a contraction event and (c) the gray edges will apply along a splitting edge, at the marker vertices present along it. Here keeping one edge rigid, resp. deforming it slightly, forces its aligned pair to have the same behavior; moreover, the deformation of the dihedral angle of an edge forces the flatness (and hence, rigidity) of the perpendicular pair. This process (of inferring rigidity/flatness of an edge from what happens with another edge incident to the same vertex) will be called *rigidity transport*. We summarize these very simple facts in the following Lemma.

Lemma 8.4.1. *The rigidity/flexibility dependency patterns from Fig. 8.3 correctly depict* 4*-vertex origami configurations in the vicinity of the flat, open state.*

Proof. The proof relies on a calculation made by Bricard in his famous memoir ([18]) on flexible octohedra; see [19] for a modern English translation. What in [19] is termed a *tetrahedral angle* is, in our terms, a 4-edged single-vertex origami, and can further be viewed as a spherical four-bar mechanism. A *face angle* is the interior angle of the central vertex, and a *dihedral angle* is the angle between the planes supporting two adjacent faces. Bricard's memoir has two parts. The first analyzes the relationship between adjacent dihedral angles of a spherical four-bar mechanism. The second is his analysis of flexible octohedra, which we do not use. Bricard categorizes the spherical four-bar mechanisms into three types based on relationships between the face angles. All of our gadgets are of Bricard *Type 3b*, which comprises those 4-edged single-vertex origami in which opposite angles are supplementary.

Given a gadget from Fig. 8.3, let face angles $\alpha, \beta, \gamma = \pi - \alpha, \delta = \pi - \gamma$ be given in (cw or ccw) order and let ϕ and ψ be the dihedral angles between the two faces spanning the (α, β), resp. (β, γ) pairs of angles. The angle labels are illustrated in Fig. 8.3(b, c, e, f). The dihedral angle values are taken between 0 and π. Let $t = \tan(\phi/2)$ and $u = \tan(\psi/2)$. Bricard showed that for *Type 3b* structures:

(1) opposite dihedrals have equal angle measure, and

(2) the parameters t and u satisfy the quadratic equation:

$$sin(\alpha + \beta)t^2 + 2\sin\beta tu - \sin(\alpha - \beta)u^2 = 0$$

Equation (2) is valid unless ϕ or ψ are π. In that case we need a different derivation. If instead we define t and u by $t = \cot(\phi/2)$ and $u = \tan(\psi/2)$ and following Bricard's derivation we obtain:

(2') the parameters t' and u' satisfy the quadratic equation:

$$sin(\alpha - \beta)t^2 + 2\sin\beta tu - \sin(\alpha + \beta)u^2 = 0$$

In Bricard's derivation there appear $\tan\beta$ and $\tan\delta$ terms, which mean for cases (a) and (b), the derivation does not work since the tangent is undefined for $\beta = \delta = \pi/2$. To prove cases (a) and (b) we use elementary spherical geometry. To prove (c)-(h) we use (1) and (2') above. The rigidity implications of (f) and (h) follow immediately from property (1). We next prove cases (c-e) and (g) using (2'). We then prove cases (a) and (b) with elementary spherical geometry.

In the **black** and **white** cases (c-e) and (g), we have the sum of the opposite angles $\alpha + \gamma = \beta + \delta = \pi$. Assume without loss of generality that $0 < \beta \leq \alpha < \pi$ and $\alpha + \beta \neq \pi$. In this case the coefficients in equations (2') are not zero. A simple analysis of these two equations shows that $t = 0$ if and only if $u = 0$. This completes the proof of all four cases.

We prove the **gray** cases (a) and (b) using the corresponding spherical four-bar linkage *abcd* on the unit sphere where a corresponds to the "input" crease (See Fig. 8.3(right)). The length of each bar is $\pi/2$ and the interior angle of each corner is equal to the dihedral of the corresponding crease. In case (a), $\angle a$ is π and so b and d are antipodal. By elementary spherical trigonometry, $\angle c$ must be π as well. In (b) we assume that $\angle a$ is between 0 and π. We add a bar bd to form spherical triangles abd and cbd. By elementary spherical trigonometry it follows that $\angle b$ and $\angle d$ in abd and cbd must be $\pi/2$ completing the proof. □

We are now ready to prove Theorem 8.3.1.

Theorem 8.3.1. (Rigid Universal Molecules) *There exist universal molecules which are rigid in the open-flat state.*

Proof. Overview: We prove existence by showing that the universal molecule crease pattern in Fig. 8.1 is rigid. The proof is by contradiction: we assume that the crease pattern is flexible and derive a contradiction by analyzing a potential nearby realization, in which the dihedral angle of at least one edge must be (slightly) smaller than 180^o. We use the types of folds that may occur at vertices of degree 4, as classified in Lemma 8.4.1. We will start at the central vertex and "propagate" flat (rigid) edges and flexed edges by sequentially applying one of the inferences (input-implies-output) proven in Lemma 1 and illustrated in Fig. 8.3. If an edge incident to a vertex is rigid, the degree of the vertex is reduced by one, when its flexibility is analyzed. A step in such a sequence of inferences is illustrated in Fig. 8.4.

The analysis of flexibility of our example reduces to just two cases. *First,* we assume that at the most central vertex of the crease pattern, all edges remain flat. Using Lemma 8.4.1, we then iteratively propagate the "flatness" and infer that all edges must remain flat. *Otherwise,* one of the edges incident to the central vertex is not flat, i.e. either a valley or a mountain. Following again the simple rules of local

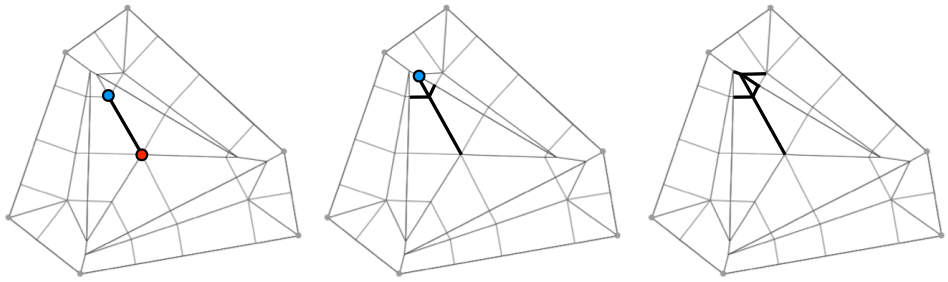


Figure 8.4: An example of the logical inference used to prove rigidity (Case 1). The red vertex at the center is assumed to be rigid. We analyze the blue highlighted vertex in each figure and conclude, from its type and rigidity of one incident edge, the rigidity of its neighbors. We continue this process for all vertices and conclude that the entire origami must be flat.

foldability at neighboring vertices, we arrive at a contradiction where one edge will need to be both flat, and not flat, simultaneously. From this we conclude that the crease pattern in Fig. 8.1 should be rigid.

We analyze now in detail each case, using the guidelines from Fig. 8.5: (left) vertex labels, (center) vertex types (black, gray and white, as classified in Lemma 8.4.1 and illustrated in Fig. 8.3) and (right) an oriented inference "path" leading to a contradiction, in Case 2 below.

Case 1: all edges incident to 1 are flat. The following inference (see Fig. 8.4) show that, in this case, all edges of the crease pattern have to be flat: the vertices 2, 6, 10, incident to 1, are white, have a rigid input edge (the one coming from 1), hence the other edges (to 5, 7, 18 etc.) are implied to be rigid; 13, 18, 22 are black each with a flat incident edge, and thus all edges incident to them are flat. Therefore, 17, 23, 28 (also black) are flat. Then the gray 3, 5, 7, 9, 12, and 24 are incident to two flat edges incident to the same face and are thus flat. By the same reasoning, 4, 8 and 11 are flat; 16, 21, and 27 are white, with one flat edge, hence all are flat; 15, 20, 26 are flat by the same reasoning; 14, 19, 25 now have all but 3 edges proved to be

flat by the statements above; since none of these are collinear, they must be all flat. Thus all edges are flat: contradiction.

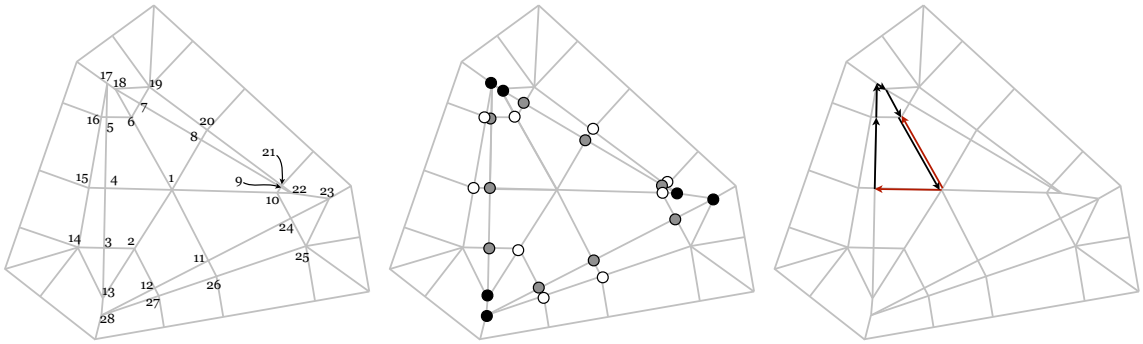


Figure 8.5: Illustration of the methodology for deriving a contradiction from the assumption that this crease pattern flexes.

Case 2: some edge incident to vertex 1 is not flat. In this case, at least 4 of them must be non flat, and hence in one of the 6 pairs of consecutive edges, neither edge is flat. A contradiction will be derived in each case, and all cases are similar, so we present the argument only for the pair of edges (1,4) and (1,6), assumed to be displaced (not flat), as in Fig. 8.5(right). Since (1, 4) is not flat and vertex 4 is gray, edge (4, 5) is flat; since (4, 5) is flat, and 5 is gray, then (5, 17) is flat; (5, 17) flat and 17 black, implied that (17, 18) is flat; (17, 18) being flat implies that (18, 6) is flat. Finally, (18, 6) being flat implies by (b) that (6, 1) is flat. This contradicts the assumption that (1, 6) is deformed away from flatness. What completes the proof is the observation that the same sequence of inferences applies to all possible subsets of edges incident to 1. □

8.5 Crease Patterns with Isolated Peaks

We extract now the characteristic features of the generic rigid example from the previous section to obtain the following generalization. The distinguishing feature of the degree-6 vertex in the example used to prove Thm. 8.3.1 is that each edge incident to it is also incident to a splitting edge and not a boundary edge. We call such a vertex

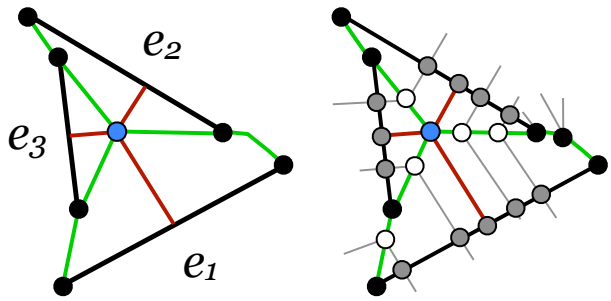


Figure 8.6: The crease pattern induced by an isolated peak in the splitting graph.

an *isolated peak* of the crease pattern. A degree-6 isolated peak occurs if and only if the splitting graph has an internal triangular face. We now extend the result of the previous section to all generic universal molecule crease patterns with an isolated peak.

Theorem 8.3.2. (Universal Molecules with isolated peaks are rigid) *If a generic universal molecule crease pattern has an isolated peak, then it is rigid in the open-flat state.*

Proof. The intuition behind the proof comes from the following observation: the example shown to be rigid by Theorem 8.3.1 contains a special vertex "surrounded" by three split edges. On the other hand, the leftmost molecule from Fig. 8.2 doesn't have such a vertex. We use now the splitting graph, defined previously as an outer-planar graph whose outer cycle corresponds to the given polygon and the diagonals correspond to the splitting events. Then we apply the argument used in the proof of Theorem 8.3.1 to obtain a sufficient condition for the universal molecule crease pattern to be rigid.

The proof requires an understanding of Lang's algorithm and of its properties from Ch. 5. We follow the algorithm as it identifies the three splitting edges making an isolated face in the splitting graph. These edges e_1, e_2, e_3 are added to the Universal Molecule crease pattern at events happening at different heights $h_1 < h_2 < h_3$ (by

the assumption of genericity). Then, the splitting edge e_1 is on the h_1-contour, and includes edge e_2 and its h_2-contour, which in turn includes edge e_3 and its h_3-contour.

Extending the crease pattern with bisector and perpendiculars around these three splitting edges, we obtain a pattern illustrated in Fig. 8.6. On the left, we see the three splitting edges (black) and their endpoints (of "black"-type, according to the classification from Lemma 8.4.1). There is a unique center vertex (the "peak") of degree 6, with three green bisectors and three red perpendiculars emanating from it. The other endpoints of the green bisectors are exactly as they are depicted in the picture: two go to edge e_3, the latest to be added as a split edge, and one goes to e_2. The endpoints of the splitting edges are connected by a path of bisector edges, and additional vertices of the crease pattern may be present along all these segments, as illustrated in Fig. 8.6(right).

With this pattern in place, one recognizes immediately the applicability of the proof of Theorem 8.3.1 to derive that all the edges that are part of this figure (of three splitting segments and all of their incident edges) must be rigid. To complete the proof for the entire crease pattern, we proceed by induction. First, we identify a few properties of the Universal Molecule crease pattern (which follow from the invariants of the algorithm). The base case of a generic recursive call to Lang's algorithm is when the contour polygon is a triangle. The bisectors of the triangle meet in one vertex, which we'll call a peak; indeed, in the Lang base state, these will be points of local maximum height for the folded paper. Generically, these are the only vertices of degree larger than 4 (namely, 6) that appear in the crease pattern. Next, we remark that each splitting segment has exactly one peak vertex on each side, and each peak is connected by two paths of bisector segments, to the endpoints of each split segment in its vicinity and by a path of perpendicular edges to some point on such a splitting edge. Therefore, if a splitting edge is proven to be "rigid" because of what happens on one of its sides, then all the edges incident to it are so, and the rigidity is transported to the peak

on the other side. To complete the proof, we show inductively (proceeding outwards from an inner triangle towards the polygon sides) that all the split edges become rigid, once those of an isolated triangle (in the split graph) have been proven to be so. □

8.6 Stable Lang Bases

We now extend the previous arguments to prove that not only is the flat state inflexible, but the folded Lang base state produced by Lang's algorithm is *stable* and cannot be unfolded. This requires first some conceptual clarifications.

The Lang base state. In this state, the *perpendicular* creases of the universal molecule are grouped together, overlapping in groups that project to internal nodes of the input metric tree. They act as hinges about which *flaps* can be rotated. The remaining creases are folded completely as either mountain or valley folds.

The Lang base state is therefore flexible, suggesting that it may be able to reach other interesting configurations through appropriate deformations. Two faces sharing creases that are bisectors or splitting edges are folded flat, one on top of the other, while the flaps made of faces sharing a perpendicular edge will have a rotation motion. Fig. 8.7, showing from below a slight perturbation of a Lang base (just enough to see which faces overlap and which not) may help with visualizing these properties. A state which is obtained from the flat Lang base simply by rotating the flaps about their incident hinges is said to be *tree-reachable.*

We prove now that the Lang base state may sometimes be stable, or *not unfoldable*, meaning that there is no nearby configuration which is not tree-reachable. To be *unfoldable*, i.e. to unfold, requires the bisector and splitting edges to *open* slightly. Our goal is to show that no such crease is opening, i.e. it cannot have a non-zero dihedral angle (while maintaining rigidly the faces) in a small neighborhood of some tree-reachable configuration.

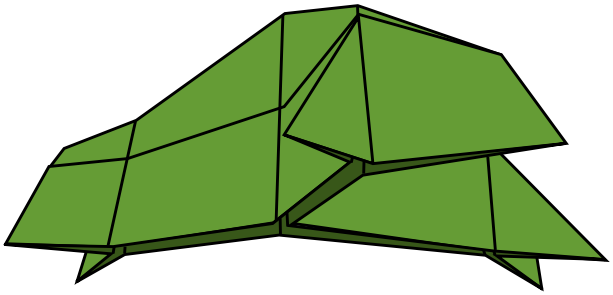


Figure 8.7: A Lang base (view from below), visualized by slightly perturbing the metric while maintaining the combinatorial structure of the realization, so that its folding pattern can be seen.

Theorem 8.3.3. (Stability of Lang Bases with isolated peaks) *All rigid folding motions of a Lang surface correspond to motions of the underlying tree. In other words, the Lang surface cannot be "unfolded" all of its faces remain perpendicular to a common plane throughout any rigid folding motion.*

Proof. We follow a similar plan as for Theorem 8.3.2. The critical step is the base case, i.e. the counterpart of Theorem 8.3.1, which relies on a slightly different set of gadgets. These, and the chain of implications leading to a contradiction to the assumption that the base is not stable in one of two cases, are depicted in Fig. 8.8. The generalization to those cases where the splitting graph has internal triangles is the same as in Theorem 8.3.2, therefore we focus now on the base case.

Rigidity Transport. In this case we assume that a nearby state exists where some non-perpendicular crease is opening and analyze the rigidity transport on the graph. Instead of transporting "flatness" of an edge, we transport the property of being "closed" or "slightly open". The gadgets for this transport are shown in Fig. 8.8 (left).

The analysis of these gadgets (a)-(f) of Fig. 8.8 essentially follows the same reasoning as the proof of Lemma 8.4.1. We use the same results of Bricard listed there. The transport cases illustrated in Figure 8.8 (c), and (d) follow directly from (1). The proof of (a) follows from the observation that if $\angle a = 0$ in the spherical four-bar

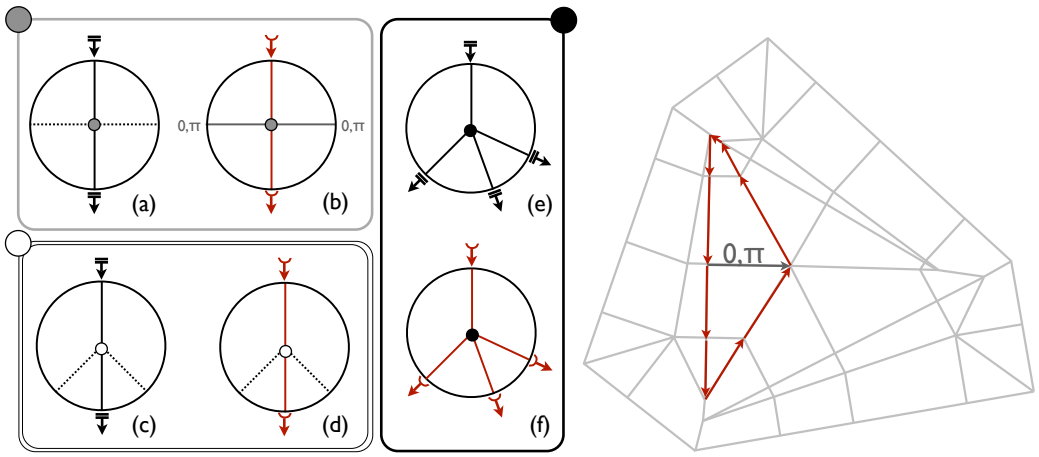


Figure 8.8: The gadgets used in the proof of Theorem 8.3.3, and the chain of implications in one of two cases needed to contradict the hypothesis that an unfolding of the Lang base exists. Arrows indicate input and output edge behavior. One with two bars indicates a "folded" (i.e. dihedral angle zero) edge. One with a crescent indicates an "opened" (i.e. dihedral angle slightly larger than zero) edge. Dotted edges are undetermined and gray edges are a special type in which the dihedral may be only either 0 or 180^o.

linkage *abcd* induced by the origami, then $b = d$ and the spherical triangle *bcd* has a zero-length edge *bd*. The proof of the transport of (b) is the same as the proof of the transport of (b) in Fig. 8.3. The proof of the transport of (e) and (f) is exactly the same as for (c) and (d) of Fig. 8.3 except using equation (2) rather than (2').

We now complete the proof of Theorem 8.3.3. To obtain a contradiction, we assume that a nearby state exists and proceed in two cases: **1.** all non-perpendiculars incident to 1 are closed, and **2.** one of the non-perpendiculars is opening. In each case we derive a contradiction. We analyze in detail each case, using the vertex labels (left) and vertex types (center) from Fig. 8.5, and the oriented inference "path" from Fig. 8.8 (right) leading to a contradiction in Case 2 below.

Since (1, 6) is opening and 6 is white, (6, 18) is opening. Since (6, 18) is opening, and 18 is black, (18, 17) is opening. Similarly, (17, 5) is opening. (17, 5) is opening, and 5 is gray, so (5, 4) is opening. (5, 4) is opening, and 4 is gray so (4, 3) is opening, *and* (4, 1) is either 0 or 180^o. (4, 3) is opening, and 3 is gray, so (3, 13) is opening. (3, 13) is opening, and 13 is black, so (13, 2) is opening. (13, 2) is opening, and 2 is white

so (2, 1) is opening. To complete the proof, we observe that an analogous sequence beginning with (1, 2) shows that (1, 11) is 0 or 180^o and (1, 10) is opening. Then the same sequence beginning at (1, 10) shows (1, 8) is 0 or 180^o. This does not depend on which non-perpendicular of 1 we begin with and no such state exists for vertex 1. □

8.7 Conclusion

For the family of crease patterns generated by Lang's Universal Molecule algorithm, in which the outerplanar splitting graph has an isolated peak, we have proven that the final Lang base state may be reachable by continuous flat-faced folding from the initial flat state. Even stronger, we showed that the initial, creased paper does not move at all if the faces are to remain rigid. We also proved that for the same crease patterns, the folded Lang base state cannot be unfolded.

Fig. 8.2 shows an example of a metric tree and two possible Lang molecules for it, whose splitting graphs indicate that one is rigid. The flat-face flexibility of the other, if true, will have to be established by other means. Thus, a full characterization of the flexible origami patterns produced by Lang's algorithm remains an open question, and Lang's algorithm requires further investigation as to which crease patterns yield continuously foldable origamis.

CHAPTER 9

CONCLUSION

In this dissertation we have investigated mathematical and algorithmic properties related to two skeleton structures that have applications to computational origami, the straight skeleton and the universal molecule. We have given faster algorithms for computing both the straight skeleton (Ch. 4) and the universal molecule (Ch. 6). We gave the first proof of correctness for the universal molecule algorithm and the first full characterization of the family of surfaces that are folded from universal molecules (Ch. 5). Using this characterization, we then generalized the universal molecule to non-convex polygons, removing a main failure mode of Lang's TreeMaker method for origami design (Ch. 7) and proving a conjecture of Demaine and O'Rourke. Finally, we showed that though the universal molecules are rigidly foldable in the rare event that the universal molecule backbone is precisely the straight skeleton, there exists a large family of universal molecules which are not foldable, and indeed are completely rigid in the open, flat state (Ch. 8). Much of this work has previously appeared in [13, 17, 14, 15]. The results of Chs. 4 & 7 have been submitted and are under review [12, 16].

Future work. Before the work in this thesis, the fastest algorithms for computing the straight skeleton of a PSLG did not use the motorcycle graph as input. Our algorithm brings the computation of the straight skeleton of a PSLG in line with that of a polygon, namely that the computational bottleneck shifts to the computation of the motorcycle graph. Any improvement in the running time of the motorcycle graph computation, then, improves the running time of our straight skeleton algorithm. Thus a major open question is how fast can the motorcycle graph be computed.

In regards to the universal molecule, this work removes one of the main failure modes of TreeMaker–when TreeMaker produces non-convex polygons, it can use our generalization of the universal molecule to produce a crease pattern. This is an improvement, but it remains the case that the first phase of TreeMaker is necessarily a heuristic, since it is solving an NP-hard problem. Thus, an interesting avenue of future work is to replace the first phase of TreeMaker with some other method, preferably something that can be solved in polynomial time. Another interesting open problem is further generalizations of the universal molecule to more exotic forms. For instance, though store-bought paper is flat, many origamists make their own paper, and it is possible to produce paper with varying curvature. An interesting question, then, is how to generalize the universal molecule to non-flat papers, and what sort of additional surfaces can be folded once this is removed.

Finally, we have only partially solved the foldability problem. We have shown that any generic universal molecule crease pattern with an isolated peak is not-foldable and it was previously known that universal molecules that are equivalent to the straight skeleton are foldable; however, there remain universal molecule crease patterns that do not fit in either of these categories. We leave it as an open problem the question of a full categorization the universal molecules.

BIBLIOGRAPHY

[1] Robert j. lang origami. http://www.langorigami.com. Accessed: 2014-08-14.

[2] Agarwal, Pankaj K., Kaplan, Haim, and Sharir, Micha. Kinetic and dynamic data structures for closest pair and all nearest neighbors. *ACM Trans. Algorithms 5*, 1 (2008), 4:1–4:37.

[3] Aichholzer, O., and Aurenhammer, F. Straight skeletons for general polygonal figures in the plane. In *Voronoi's Impact on Modern Sciences II*, A.M. Samoilenko, Ed., vol. 21. Proc. Institute of Mathematics of the National Academy of Sciences of Ukraine, Kiev, Ukraine, 1998, pp. 7–21.

[4] Aichholzer, Oswin, Alberts, D., Aurenhammer, Franz, and Gärtner, Bernd. A novel type of skeleton for polygons. *Journal of Universal Computer Science 1*, 12 (1995), 752–761.

[5] Aurenhammer, Franz. Voronoi diagrams – a survey of a fundamental geometric data structure. *bit acm computing surveys '91 (Japanese translation), Kyoritsu Shuppan Co., Ltd.* (1993), 131–185.

[6] Barequet, Gill, Goodrich, Michael T., Levi-Steiner, Aya, and Steiner, Dvir. Straight-skeleton based contour interpolation. In *Proc. 14th Symp. on Discrete Algorithms* (Philadelphia, PA, USA, 2003), SODA '03, Society for Industrial and Applied Mathematics, pp. 119–127.

[7] Basch, Julien, Guibas, Leonidas J., and Hershberger, John. Data structures for mobile data. In *Proc. 8th ACM-SIAM Symp. Discr. Algorithms (SODA)* (1997), SODA '97, pp. 747–756.

[8] Bern, Marshall Wayne, and Hayes, Barry. The complexity of flat origami. In *SODA: ACM-SIAM Symposium on Discrete Algorithms* (1996).

[9] Biedl, Therese, Held, Martin, Huber, Stefan, Kaaser, Dominik, and Palfrader, Peter. A simple algorithm for computing positively weighted straight skeletons of monotone polygons. *Information Processing Letters 115*, 2 (2015), 243 – 247.

[10] Blum, Harry. A transformation for extracting new descriptors of shape. In *Proc. Models for the Perception of Speech and Visual Form* (Cambridge, MA, November 1967), Weiant Wathen-Dunn, Ed., MIT Press, pp. 362–380.

[11] Bowers, John C. Computing the straight skeleton of a simple polygon from its motorcycle graph in deterministic o(n log n) time. *CoRR abs/1405.6260* (2014).

[12] Bowers, John C. Faster reductions for straight skeletons to motorcycle graphs. Tech. rep., University of Massachusetts and Smith College, 2015. Submitted, Apr. 2015.

[13] Bowers, John C., and Streinu, Ileana. Lang's universal molecule algorithm (video). *Proc. 28th Symp. Computational Geometry (SoCG'12)* (2012).

[14] Bowers, John C., and Streinu, Ileana. Computing origami universal molecules with cyclic tournament forests. In *Proc. 15th Intern. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'13)* (2013), IEEE, pp. 42–52.

[15] Bowers, John C., and Streinu, Ileana. Lang's universal molecule algorithm. *Annals of Mathematics and Artificial Intelligence* (2014), 1–30.

[16] Bowers, John C., and Streinu, Ileana. Geodesic universal molecules. Tech. rep., University of Massachusetts and Smith College, 2015. Submitted, Apr. 2015.

[17] Bowers, John Christopher, and Streinu, Ileana. Rigidity of origami universal molecules. In *Automated Deduction in Geometry* (2012), Tetsuo Ida and Jacques D. Fleuriot, Eds., vol. 7993 of *Lecture Notes in Computer Science*, Springer, pp. 120–142.

[18] Bricard, Raoul. Mémoire sur la théorie de l'octaèdre articulé. *J. Math. Pure et Appl. 5* (1897), 113–148.

[19] Bricard, Raoul. Memoir on the theory of the articulated octahedron. Translation from the French original of [18]., March 2012.

[20] Chazelle, Bernard. Cutting hyperplanes for divide-and-conquer. *Discrete & Computational Geometry 9*, 1 (1993), 145–158.

[21] Cheng, Howard, Devadoss, Satyan L, Li, Brian, and Risteski, Andrej. Skeletal Rigidity of Phylogenetic Trees. *arXiv.org* (Mar. 2012), 5782.

[22] Cheng, S. W., and Vigneron, Antoine. Motorcycle graphs and straight skeletons. In *Proc. 13th Symp. on Discrete algorithms* (Philadelphia, PA, USA, 2002), SODA '02, Society for Industrial and Applied Mathematics, pp. 156–165.

[23] Cheng, S. W., and Vigneron, Antoine. Motorcycle graphs and straight skeletons. *Algorithmica 47*, 2 (January 2007), 159–182.

[24] Cheng, Siu-Wing, Mencel, Liam, and Vigneron, Antoine. A faster algorithm for computing straight skeletons. In *Proc. 22nd Euro. Symp. on Algorithms (ESA 2014), to appear* (Worclaw, Poland, Sept. 2014).

[25] Chin, Francis, Snoeyink, Jack, and Wang, Cao An. Finding the medial axis of a simple polygon in linear time. In *Discrete Comput. Geom* (1995), Springer-Verlag, pp. 382–391.

[26] Cloppet, Florence, Stamon, George, and Oliva, Jean-Michel. Angular bisector network, a simplified generalized voronoi diagram: Application to processing complex intersections in biomedical images. *IEEE Trans. Pattern Anal. Mach. Intell. 22*, 1 (Jan. 2000), 120–128.

[27] Demaine, Erik D., and Demaine, Martin L. Computing extreme origami bases. Tech. Rep. CS-97-22, Department of Computer Science, University of Waterloo, May 1997.

[28] Demaine, Erik D., Demaine, Martin L., and Lubiw, Anna. Folding and One Straight Cut Suffices. *Proc. 10th Annual ACM-SIAM Sympos. Discrete Alg. (SODA '99)* (Jan. 1999), 891–892.

[29] Demaine, Erik D., Demaine, Martin L., and Mitchell, Joseph S. B. Folding flat silhouettes and wrapping polyhedral packages: New results in computational origami. In *Computational Geometry: Theory and Applications* (1999), pp. 105–114.

[30] Demaine, Erik D., Fekete, Sándor P., and Lang, Robert J. Circle packing for origami design is hard. In *Origami 5: Fifth International Meeting of Origami Science, Mathematics, and Education*, Patsy Wang-Iverson, Robert J. Lang, and Mark Yim, Eds. Taylor and Francis, 2011, pp. 609–626.

[31] Demaine, Erik D., and O'Rourke, Joseph. *Geometric Folding Algorithms: Linkages, Origami, and Polyhedra*. Cambridge University Press, 2007.

[32] Edelsbrunner, Herbert. The upper envelope of piecewise linear functions: Tight bounds on the number of faces. *Discrete & Computational Geometry 4*, 1 (1989), 337–343.

[33] Eppstein, D., and Erickson, J. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. *Discr. & Comput. Geom. 22*, 4 (1999), 569–592.

[34] Farber, Michael. *Invitation to Topological Robotics*. Zürich Lectures in Advanced Mathematics. European Mathematical Society, 2008.

[35] Guibas, Leonidas J., and Stolfi, Jorge. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM T. Graphic. 4* (1985), 74–123.

[36] Hershberger, J. Finding the upper envelope of n line segments in o(n log n) time. *Inf. Process. Lett. 33*, 4 (Dec. 1989), 169–174.

[37] Huber, Stefan. *Computing Straight Skeletons and Motorcycle Graphs: Theory and Practice*. PhD thesis, Universität Salzburg, Austria, June 2011.

[38] Huber, Stefan, and Held, Martin. Computing straight skeletons of planar straight-line graphs based on motorcycle graphs. 187–190.

[39] Huber, Stefan, and Held, Martin. Theoretical and practical results on straight skeletons of planar straight-line graphs. In *Proc. 27th Symp. on Computational geometry* (New York, NY, USA, 2011), SoCG '11, ACM, pp. 171–178.

[40] Huber, Stefan, and Held, Martin. A fast straight-skeleton algorithm based on generalized motorcycle graphs. *Int. J. Comput. Geometry Appl. 22*, 5 (2012), 471–.

[41] Huzita, H. Axiomatic development of origami geometry. In *Proc. First Intern. Meeting of Origami Science and Technology (1989)*, H. Huzita, Ed. 1991, pp. 143–158.

[42] Ida, Tetsuo, Takahashi, Hidekazu, Marin, Mircea, Ghourabi, Fadoua, and Kasem, Asem. Computational construction of a maximum equilateral triangle inscribed in an origami. In *Mathematical Software - ICMS 2006*, Andrs Iglesias and Nobuki Takayama, Eds., vol. 4151 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 361–372.

[43] Ida, Tetsuo, Tepeneu, Dorin, Buchberger, Bruno, and Robu, Judit. Proving and constraint solving in computational origami. In *Artificial Intelligence and Symbolic Computation*, Bruno Buchberger and John Campbell, Eds., vol. 3249 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 132–142.

[44] Lang, Robert J. A computational algorithm for origami design. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry* (1996), pp. 98–105.

[45] Lang, Robert J. Treemaker 4.0: A program for origami design, http://www.langorigami.com, 1998.

[46] Lang, Robert J. *Origami design secrets: mathematical methods for an ancient art*. Ak Peters Series. A.K. Peters, 2003.

[47] Lang, Robert J., Ed. *Origami 4: Fourth International Meeting of Origami Science, Mathematics, and Education*. A. K. Peters, 2009.

[48] Lang, Robert J., and Demaine, Erik D. Facet ordering and crease assignment in uniaxial bases. In *[47]*. 2009.

[49] Panina, Gaiane, and Streinu, Ileana. Flattening single-vertex origami: the non-expansive case. *Computational Geometry: Theory and Applications 46*, 8 (October 2010), 678–687.

[50] Streinu, Ileana, and Whiteley, Walter. Single-vertex origami and spherical expansive motions. In *Proc. Japan Conf. Discrete and Computational Geometry (JCDCG 2004)* (Tokai University, Tokyo, 2005), Jin Akiyama and M. Kano, Eds., vol. 3742 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 161–173.

[51] Tachi, T. Simulation of rigid origami. In *Origami 4: Fourth International Meeting of Origami Science, Mathematics, and Education*, Robert J. Lang, Ed. A K Peters, 2009, ch. 3, pp. 175–188.

[52] Tachi, T. Origamizing polyhedral surfaces. *Visualization and Computer Graphics, IEEE Transactions on 16*, 2 (2010), 298–311.

[53] Tachi, Tomohiro. Generalization of rigid foldable quadrilateral mesh origami. In *Proc. Int. Assoc. for Shell and Spatial Structures (IASS)* (Philadelphia, PA, USA, 2009), IASS 2009, Society for Industrial and Applied Mathematics, pp. 156–165.

[54] Tachi, Tomohiro. Freeform rigid-foldable structure using bidirectionally flat-foldable planar quadrilateral mesh. In *Advances in Architectural Geometry 2010*, Cristiano Ceccato, Lars Hesselgren, Mark Pauly, Helmut Pottmann, and Johannes Wallner, Eds. Springer Vienna, 2010, pp. 87–102.

[55] Tanase, Mirela, and Veltkamp, Remco C. Polygon decomposition based on the straight line skeleton. In *Proc. 19th Symp. on Computational Geometry* (New York, NY, USA, 2003), SCG '03, ACM, pp. 58–67.

[56] Trofimov, Vadim, and Vyatkina, Kira. Linear axis for general polygons: properties and computation. 122–135.

[57] Vanegas, Carlos A., Kelly, Tom, Weber, Basil, Halatsch, Jan, Aliaga, Daniel G., and Müller, Pascal. Procedural generation of parcels in urban modeling. *Comp. Graph. Forum 31*, 2pt3 (May 2012), 681–690.

[58] Vigneron, Antoine, and Yan, Lie. A faster algorithm for computing motorcycle graphs. In *Proc. 29th Symp. on Computational geometry* (New York, NY, USA, 2013), SoCG '13, ACM, pp. 17–26.

[59] Wang-Iverson, Patsy, Lang, Robert J., and Yim, M. *Origami 5: Fifth International Meeting of Origami Science, Mathematics, and Education*. Taylor and Francis, 2011.