Computing

Straight Skeletons

by Means of

Kinetic Triangulations

Masterarbeit

zur Erlangung des akademischen Grades Diplom-Ingenieur
an der Naturwissenschaftlichen Fakultät
der Paris Lodron Universität Salzburg

Eingereicht von Peter Palfrader

Gutachter: Ao. Univ.-Prof. Dipl.-Ing. Dr. Martin Held
Fachbereich Computerwissenschaften

Salzburg, September 2013

Computing

Straight Skeletons

by Means of

Kinetic Triangulations

Master's Thesis

Peter Palfrader

September 2013

Department of Computer Sciences
University of Salzburg
Jakob-Haringer-Straße 2
5020 Salzburg
Austria

Peter Palfrader

[commit 3942e30; September 12, 2013]

ABSTRACT

The straight skeleton is a geometric object defined for polygons, or, more generally, planar straight-line graphs (PSLGs). It was introduced to computational geometry in 1995 by Aichholzer, Aurenhammer, Alberts, and Gärtner. Roughly, a straight skeleton is a planar straight-line graph motivated by a particular shrinking process of a polygon. It introduces a partition similar to that of a Voronoi diagram.

We extensively study Aichholzer and Aurenhammer's 1998 kinetic triangulation-based algorithm to construct the straight skeleton for PSLGs. In particular, we establish that their algorithm is not properly defined for input that is not in general position. We introduce our extensions which are required so that the algorithm works correctly and terminates for all input — even when using only finite-precision arithmetic.

We also present our implementation of the refined algorithm, Surfer, and report on our extensive experiments. We provide strong experimental evidence that the number of flip events is linear in practice; the theoretical bound is in $\mathcal{O}(n^3)$. Our measurements further establish that for practical input the algorithm runs in approximately $\mathcal{O}(n \log n)$ time and that our implementation is clearly the fastest straight-skeleton code currently in existence, improving on Huber's implementation by a factor of 10 and on CGAL's by at least a factor linear in the input size.

This thesis builds on and extends work already presented at ESA2012, the 20th Annual European Symposium on Algorithms in Ljubljana, Slovenia (Palfrader, Held, Huber — *On Computing Straight Skeletons by Means of Kinetic Triangulations*, [PHH12]).

DEDICATION

To my family.

ACKNOWLEDGMENTS

I would like to thank my advisor, Martin Held, first of all for his introduction to the field of computational geometry and second for suggesting such an interesting topic for my thesis. He shared his wealth of experience and provided a great many good ideas for solutions to various problems I encountered along the way.

I also want to thank my colleagues and friends, in particular Dominik Kaaser and Stefan Huber, for many insightful discussions and valuable feedback during the development of my software and writing the thesis.

And last, not least, my thanks go to my parents, Agnes and Sepp, for their support, always.

TABLE OF CONTENTS

1 INTRODUCTION

The medial axis of a simple polygon is a subset of its Voronoi diagram and consists of the points of the interior whose closest point on the polygon is not unique.

The straight skeleton of a simple polygon was introduced in 1995 by Aichholzer et al. [AAAG95b]. It is a structure not unlike the medial axis. However, while the medial axis features segments of parabolas in addition to line segments, the straight skeleton consists entirely of line segments.

For reflex vertices of an input polygon the offset curves induced by the medial axis will contain circular arcs. The offset curve induced by the straight skeleton will, again, only have line segments. See Figure 1.

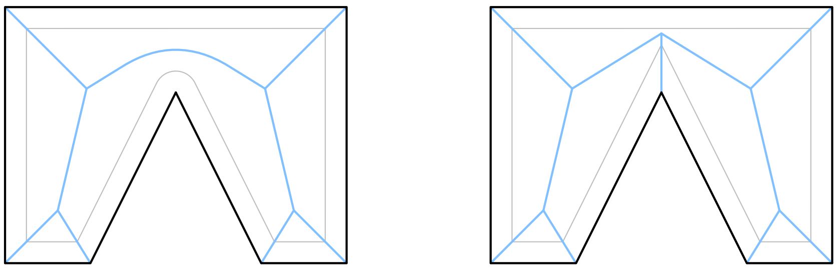
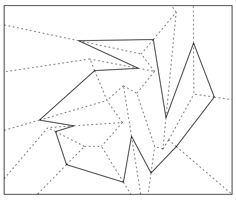


Figure 1: Medial axis of a polygon on the left in comparison to a straight skeleton on the right. Both the medial axis and the straight skeleton are shown in blue, induced offset curves in gray.

1.1 APPLICATIONS

Straight skeletons have applications in diverse fields of industry and science. They can be used to efficiently compute mitered offset curves in NC machining, using an approach similar to the construction of an offset with round corners based on the medial axis. Tomoeda et al. use the straight skeleton to create signs with an illusion of depth [TS12]. Straight skeletons have been used in mathematical origami to create fold or cut patterns or design pop-up cards [DO07, DDL98, DDM00, ADD+13, Sug13]. They can also be used in roof design [AAAG95b, LD03] and terrain generation.



1.2 CONTRIBUTION

We investigate Aichholzer and Aurenhammer's triangulation-based algorithm for constructing the straight skeleton in detail [AA98].

We first describe their algorithm and then report on the requirement of triangulating the entire plane and the technical details of handling unbounded triangles.

Furthermore, we present our extensions which are required to correctly compute the straight skeleton of a general planar straight-line graph without relying on an implicit assumption of general position of the input. In particular, we describe how to handle parallel input edges which lead to vertices moving at infinite speed in the kinetic triangulation. Additionally, we provide solutions to the problem of infinite loops during the handling of flip events. Such flip-event loops are not only of theoretical interest and concern but actually happen in practice. Our first approach requires exact arithmetic operations and describes an ordering of events which avoids flip-event loops entirely. The second approach also works with limited precision such as standard IEEE 754 double-precision floating-point operations implemented in common computing hardware.

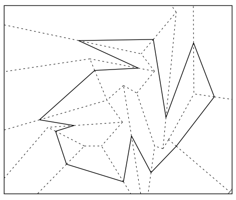
We implemented the algorithm and our extensions in C and present statistics based on our extensive test runs. In particular, we address the question raised by Aichholzer and Aurenhammer of how many flip events can be expected to occur in practice.

Our code, Surfer, is implemented as a library and supports two arithmetic back-ends, namely IEEE 754 double-precision floating-point and the extended-precision library MPFR [MPF]. While the algorithm's worst case complexity is in $\mathcal{O}(n^3 \log n)$, our tests indicate that Surfer runs in $\mathcal{O}(n \log n)$ time for all practical input. Memory consumption is, as expected, in $\mathcal{O}(n)$. Additionally, Surfer is robust and fast enough to process inputs of a few million vertices with floating-point arithmetic in a matter of seconds.

Comparing Surfer with the straight skeleton code in CGAL [CGA] shows that our implementation has several advantages: it is faster by a factor linear in the input size, it can handle arbitrary PSLGs as input and not just polygons with holes, and it requires only linear, not quadratic, space. We also compare our code with Bone by Huber and Held [HH11]. Even though their algorithm has a better worst case complexity, we observe that in practice our code is faster by a factor of approximately 10.

1.3 OUTLINE

We provide the definition of the straight skeleton and present an overview of its properties and of algorithms that compute it in Chapter 2. Aichholzer and Aurenhammer's triangulation-based algorithm is introduced in detail in Chapter 3 and we describe our implementation in Chapter 4. Chapter 5 covers problems of the original algorithm when its input is not in general position. In particular, we cover how to detect and resolve what we call *flip-event loops*. In Chapter 6 we present experimental results, including observed count of flip events and performance of our implementation. Chapter 7 summarizes our results.



2 STRAIGHT SKELETONS

2.1 DEFINITION

The straight skeleton of simple polygons was introduced by Aichholzer, Alberts, Aurenhammer, and Gärtner in 1995 [AAAG95b] by looking at a particular shrinking process of a polygon. Each edge e of a polygon P emanates a wavefront edge w towards the interior of P such that w is parallel to e and moves away at unit speed. The propagating wavefront is the set of such wavefront edges, joined in a way that the angles at which edges of P meet are preserved in the wavefront.

As the wavefront propagates, two types of topological changes can occur: (i) wavefront edges shrink to zero length and vanish, and (ii) wavefront vertices crash into an opposing wavefront edge. The former type of events are called *edge events* and the latter *split events* for they cut a single interior region of the shrinking polygon into two, or, equivalently, because they split the opposing wavefront edge into two distinct line segments.

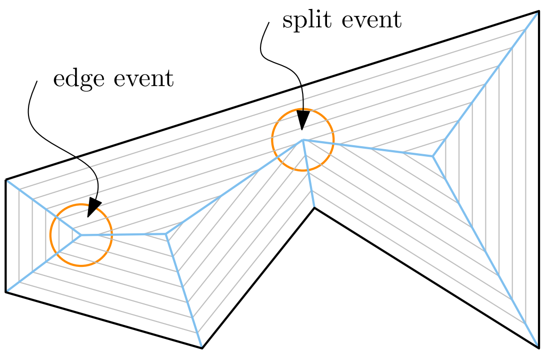
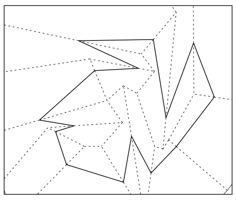


Figure 2: Edge and split events.

The process ends when the shrinking polygon or polygons have all collapsed and, accordingly, all wavefront-edges have vanished.

DEFINITION. During this wavefront propagation, the vertices of the wavefront trace out straight-line segments. The straight skeleton $\mathcal{S}(P)$ is the union of these traces.



To make it clearer whether one refers to elements of the input graph or elements of the straight skeleton, Aichholzer et al. called edges of the straight skeleton *arcs* and vertices of the straight skeleton *nodes*.

Figure 2 shows a simple input polygon P and several offset curves in gray that represent the wavefront at distinct times in the propagation process. The arcs of the straight skeleton in blue cover the traces of wavefront vertices, and its nodes witness changes in the wavefront topology. The figure furthermore highlights one edge and one split event.

MORE GENERAL INPUT. Aichholzer and Aurenhammer later extended the definition to consider a straight skeleton for any planar straight-line graph (PSLG) [AA98].

Given that there is no interior per se, edges of the input graph G emanate self-parallel wavefront edges on both of their sides. Vertices that have a degree of one, called *terminal vertices*, cause an extra wavefront to be sent out orthogonal to their incident edge.

Unlike in the case of a simple polygon, not all wavefront edges will collapse eventually. Instead, a few of them will escape to infinity causing some of the straight skeleton arcs to be rays instead of line segments.

Figure 3 shows a straight skeleton of a planar straight-line graph.

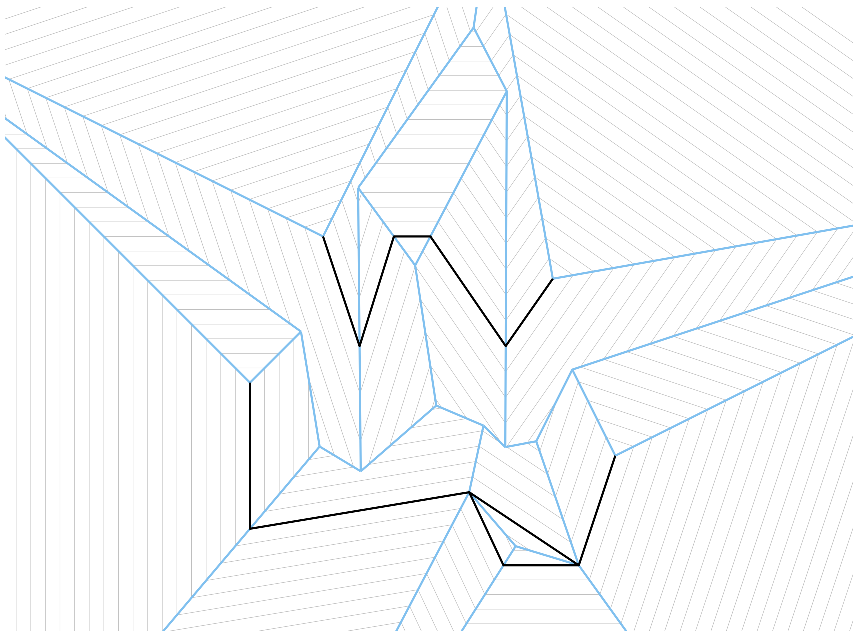


Figure 3: Straight skeleton (blue) of a PSLG (black) with offset curves (gray).

2.2 PROPERTIES OF STRAIGHT SKELETONS

LEMMA. Let G be a planar straight-line graph. The straight skeleton $\mathcal{S}(G)$ consists entirely of straight-line segments (or rays).

PROOF. The arcs of $\mathcal{S}(G)$ are the traces of wavefront vertices during the wavefront propagation process. Each wavefront vertex moves on the angular bisector of the supporting lines of two input edges. Since the angular bisector is a line, a straight skeleton arc is either a line segment or, if it does not stop and goes to infinity, a ray. $\square$

LEMMA ([AAAG95a]). Let P be a simple polygon. Each edge e of P induces one connected face in $\mathcal{S}(P) \cup P$.

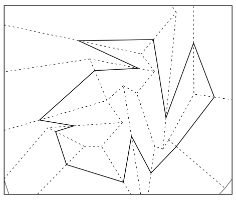
PROOF. Each edge e of P emanates one wavefront edge $w(e)$. The area this wavefront edge sweeps over is called the face of e, or $f(e)$. At the start of the wavefront propagation, $f(e)$ is equal to e. As the wavefront propagates, $w(e)$ moves towards the interior of the polygon and $f(e)$ grows accordingly. While a split event can cause $w(e)$ to split, it cannot disconnect $f(e)$. Edge events also cannot disconnect $f(e)$. Once $w(e)$ has disappeared it cannot re-appear again. Therefore, $f(e)$ is connected. $\square$

Similarly, the faces $f(e)$ in the straight skeleton of a planar straight-line graph are connected.

LEMMA ([AAAG95a]). Let P be a simple polygon with n vertices. Then $\mathcal{S}(P)$ is a tree of exactly $n-2$ nodes and $2n-3$ arcs and partitions the interior of P into exactly n connected faces.

Note that the following proof assumes that all nodes of a straight skeleton are of degree three. This can be achieved by creating a single node for each topological change. Even if two or more nodes coincide geometrically, we will not merge them and still consider them distinct nodes.

PROOF. We already established that each edge of P induces one connected face. Since P has n edges, there are n faces. Each point in the interior of P will eventually be visited by the wavefront. Therefore, $\mathcal{S}(P)$ partitions the interior of P into exactly n connected faces. Each face is bounded by arcs of $\mathcal{S}(P)$ and one edge of P. Thus, $\mathcal{S}(P)$ is a tree. This tree has n leaves and as all inner vertices have a degree of three it follows that $\mathcal{S}(P)$ consists of $n-2$ nodes and $2n-3$ arcs. $\square$



LEMMA ([AAAG95a]). Each face $f(e)$ of $\mathcal{S}(G) \cup G$ is monotone with respect to e.

PROOF. Assume there is a line l, normal to e, that at one point x leaves $f(e)$ only to re-enter it at some other point y further from e. Between x and y this line l intersects wavefront edges that are no longer parallel to e. Thus, these wavefront edges will reach y before $w(e)$ does which contradicts the definition of y. Therefore, there is no such line l and thus for every normal of e its intersection with $f(e)$ is connected. It follows that $f(e)$ is monotone with respect to e. $\square$

2.3 ROOF MODEL

Aichholzer et al. also presented a useful and interesting interpretation of the straight skeleton of polygons [AAAG95a] and planar straight line graphs [AA98]. Their 3D interpretation enables the use of the straight skeleton in roof modeling and terrain generation.

They assign to each point (x, y) of the plane a value t which corresponds to the time this point was first visited by the wavefront emanated from a graph G. The terrain of G is then the set of tuples (x, y, t) for all (x, y). Interpreting t in 3D as an elevation above the xy-plane gives rise to a sort of roof that sits on top of walls which are the edges of G at $t = 0$.

This roof has interesting properties. The slope of all faces of the roof is 1, the inverse of the propagation speed. For polygonal inputs, water running down the roof always ends up on the outside; there are no sinks for it to get trapped. A projection of the ridges and valleys of the roof onto the plane $t = 0$ corresponds to the straight skeleton.

2.4 COMPUTING THE STRAIGHT SKELETON

Unfortunately, straight skeletons are not just another instance of abstract Voronoi diagrams [AAAG95a]. Therefore, well-established algorithms to compute Voronoi diagrams cannot, in general, be used to also construct straight skeletons. Instead, computing a straight skeleton requires special-purpose algorithms. We provide an overview of such algorithms in this section.

SIMULATING THE SHRINKING PROCESS. When introducing the straight skeleton structure, Aichholzer et al. suggested that the trivial approach of simulating the shrinking process of the polygon might work well in practice [AAAG95b]. This simulation of the wavefront propagation process would work as follows: (i) Identify the first event to change the wavefront topology. (ii) Fast-forward the simulation to that event's time and update the wavefront as required. (iii) Then identify the next change of the wavefront topology. Repeat steps (ii) and (iii) until the polygon has collapsed.

To find the next edge event, consider all edges and determine for each edge e the point in time when it will intersect with its two neighbors in a single point, or, equivalently, when it will have shrunk to a length of zero. This computation of potential edge event times only needs local information, and thus can be done in $\mathcal{O}(n)$ time for a polygon.

The next split event can be found by determining, for all pairs of reflex vertices v and edges e, when v will crash into e. This requires $\mathcal{O}(r \cdot n)$ time, where r is the number of reflex vertices. Since r is in $\mathcal{O}(n)$, the time of identifying all potential split events is in $\mathcal{O}(n^2)$.

A complete shrinking process consists of $\mathcal{O}(n)$ events and therefore, if finding the next event is repeated after each step, this algorithm yields an overall complexity of $\mathcal{O}(n^3)$ time and $\mathcal{O}(n)$ space.

A change in the topology of the wavefront only affects a limited number of potential future events. Using a suitable priority queue for all event times can, therefore, reduce the run-time complexity to $\mathcal{O}(n^2 \log n)$ time at the cost of quadratic space.

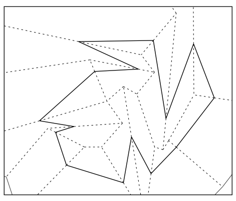
The approach using the priority queue was implemented by Cacciola for his CGAL submission [Cac12, Hub12].

FINDING INTERSECTIONS. The same year, Aichholzer et al. also explored a second approach to finding split events [AAAG95a]. They observe that a split event that is ignored will cause the polygon to become self-intersecting.

They propose to test the shrinking polygon for self-intersections after each performed edge event. Only if the test indicates that at a time t the polygon is no longer simple a split event must have been missed. Since the last event of the shrinking process will always be an edge event, their method guarantees finding all split events.

One important contribution is that they also describe how to process all split events before time t, using only the topology of the shrunk polygon at t. Testing for self-intersection can be done in linear time using Chacelle's algorithm [Cha91] and thus, if the first split event is the k^{th} event, finding it using first exponential search and then binary search can be done in $\mathcal{O}(n \log k)$ time.

Overall, this algorithm can compute the straight skeleton of a simple polygon with n vertices in $\mathcal{O}(n^2 \log n)$ time, or, if r indicates the number of reflex vertices, $\mathcal{O}(n \cdot r \log n)$ time.



KINETIC TRIANGULATION. First in 1996 and then again in 1998 Aichholzer and Aurenhammer presented a triangulation-based algorithm [AA96, AA98]. We describe this algorithm in Chapter 3 and discuss our implementation and details starting with Chapter 4.

EPPSTEIN AND ERICKSON. In their algorithm, Eppstein and Erickson use a sweep-plane approach on the roof model of a polygon P to compute its straight skeleton $\mathcal{S}(P)$ [EE99].

They embed P in the plane $z = 0$ and then set up a possibly unbounded triangle in 3D for each input edge e. The base edge of a triangle is e itself, the other two are defined to correspond to the incident upward edges in the roof model. Furthermore, for each reflex vertex v of G they create a ray supporting the valley in the roof model that starts at v.

The sweep plane is parallel to, and starts at, the plane $z = 0$. As it moves upwards, two types of events happen: (i) The sweep plane meets the top of a triangle, finishing a face of the roof. This corresponds to an edge event. (ii) The plane hits the intersection of a ray with a triangle, indicating a split event. In both cases a limited number of triangles and rays need to be removed and added to the sets the algorithm maintains.

To find the next edge event, triangles are kept in a simple priority queue. Finding the next split event is more refined and involves both ray shooting and lowest-intersection queries. To support ray shooting queries, Eppstein and Erickson use a data structure of Agarwal and Matoušek [AM94]. For lowest-intersection queries they combine fast data structures with high space complexity and slow data structures with little memory consumption to build a hierarchical structure which supports the required queries.

The result is an algorithm to compute the straight skeleton with a space and time complexity in $\mathcal{O}(n^{1+\varepsilon} + n^{8/11+\varepsilon} \cdot r^{9/11+\varepsilon})$ where r is the number of reflex input vertices and ε is a positive real. This is currently the algorithm with the best worst-case complexity.

While Eppstein and Erickson present their work only for polygonal input, their algorithm works equally for arbitrary planar straight-line graphs.

MOTORCYCLE GRAPHS. In the same paper Eppstein and Erickson furthermore distill the most difficult part of constructing straight skeletons, namely determining the fate of reflex vertices.

They consider a set of vertices, called motorcycles, on the plane. All motorcycles start moving simultaneously from their individual initial position and at their individual constant velocity. Should one motorcycle cross the track of a different motorcycle it immediately crashes and stops moving. Motorcycles that do not crash will eventually escape to infinity.

The motorcycle graph then is the set of line segments and rays that correspond to the traces of all motorcycles.

Eppstein and Erickson also describe how to compute the motorcycle graph in $\mathcal{O}(n^{17/11+\varepsilon})$ using a method similar to how they construct straight skeletons.

CHENG AND VIGNERON. Cheng and Vigneron where the first to use the motorcycle graph as a means to construct the straight skeleton of simple polygons with holes [CV02].

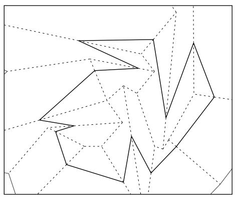
They start by presenting an algorithm to compute the motorcycle graph in $\mathcal{O}(n\sqrt{n}\log n)$ time. Cheng and Vigneron note that the number of potential crashes of n motorcycles is in $\Omega(n^2)$ but only a linear number actually occur. In order to avoid having to consider a quadratic number of crashes, they partition the plane into cells using a $1/\sqrt{n}$ cutting algorithm by Chazelle [Cha93]. This way, the only motorcycles that can cause a bike to crash are those that currently share its cell or traversed it in the past.

The authors then continue and describe how to compute the straight skeleton of a polygon P from a motorcycle graph induced by the reflex vertices of P. They employ a randomized divide and conquer approach that results in an algorithm running in $\mathcal{O}(n\log^2 n)$ expected time.

Combining both elements yields, for an input polygon with n vertices of which r are reflex, an expected runtime in $\mathcal{O}(r\sqrt{r}\log r + n\log^2 n)$.

HUBER AND HELD. Huber and Held presented another algorithm to construct the straight skeleton from the motorcycle graph [HH10a, Hub12].

Like Aichholzer et al. [AA98], they simulate the wavefront propagation process and maintain a graph of the wavefront. Their core idea is to combine this graph with those parts of the motorcycle graph that have not yet been swept over by the wavefront. This results in a kinetic graph which consists exclusively of convex faces. Any change in its topology is thus witnessed by the collapse of an edge of the graph. An edge collapse, called an event, causes some topological change in the kinetic graph. Huber and Held maintain a priority queue of event times and process them in chronological order.



Their algorithm has a worst case complexity in $\mathcal{O}(n^2 \log n)$. Extensive tests performed with their implementation suggest a runtime in $\mathcal{O}(n \log n)$ for practical applications.

3 TRIANGULATION-BASED ALGORITHM

In their 1998 paper Aichholzer and Aurenhammer [AA98] built upon their earlier work by extending the definition for the straight skeleton from polygons to planar straight-line graphs (PSLGs) and, furthermore, presented an algorithm to compute the straight skeleton of arbitrary PSLGs using a triangulation-based approach.

The algorithm simulates the wavefront propagation process over time. The core idea is to maintain, at all times, a triangulation of the part of the plane that has not yet been swept over by a wavefront edge. Each change in the wavefront topology is witnessed by the collapse of a triangle in this ever changing triangulation. Thus, by determining the time when the next triangle will collapse, one can establish when the next change in the wavefront topology will occur and advance the clock accordingly.

3.1 INITIAL WAVEFRONT

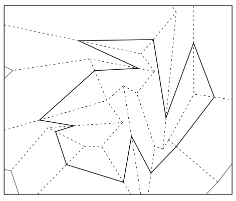
In more detail, the algorithm starts by computing the initial wavefront from a given input PSLG G. Recall that each input edge e of G emanates two wavefront edges parallel to e and that terminal vertices emanate one wavefront edge orthogonal to their incoming input edge.

The initial wavefront $\mathcal{W}(G)$ is therefore created by duplicating edges of G and creating initial zero-length edges for each terminal vertex of G. These wavefront edges are appropriately linked together by wavefront vertices, all of which have a degree of two.

Initially the vertex sets of G and $\mathcal{W}(G)$ cover the same points in the plane. The same holds for their edge sets.

Each wavefront edge w is considered to have a speed of 1, moving in a self-parallel manner. A wavefront vertex or kinetic vertex v joining two edges w_1 and w_2 therefore moves along the bisector of w_1 and w_2.

SPEED OF KINETIC VERTICES. In order to keep up with its incident wavefront edges, a wavefront vertex has to move at a certain speed. Let α be the angle between the two wavefront edges incident

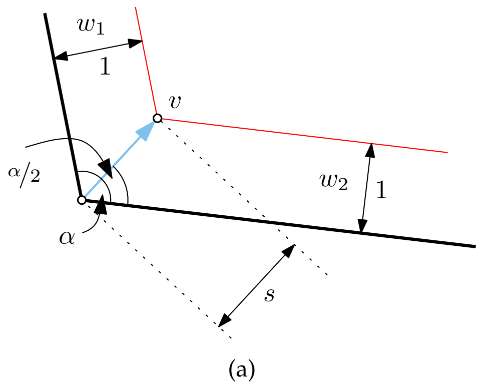


at v and let s be the speed of v. An instance where α is less than π is shown in Figure 4a. Given that the wavefront edge moves at unit speed,

$$\sin\left(\frac{\alpha}{2}\right) = \frac{1}{s}$$

holds. Solving for s yields a speed of the kinetic vertex v of

$$s = \frac{1}{\sin \alpha/2}.$$



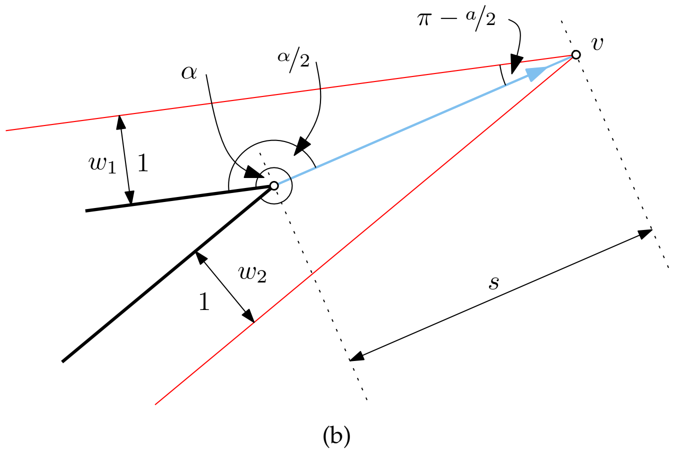


Figure 4: A kinetic vertex v moves on the bisector between its incident wavefront edges at a speed dictated by the angle between its incident wavefront edges.

If the angle between wavefront edges is larger than π, as depicted in Figure 4b, then

$$\sin\left(\pi - \frac{\alpha}{2}\right) = \frac{1}{s}.$$

Since $\sin(\gamma) = \sin(\pi - \gamma)$ for all angles γ, this actually solves to the same speed for v, so again

$$s = \frac{1}{\sin \alpha/2}.$$

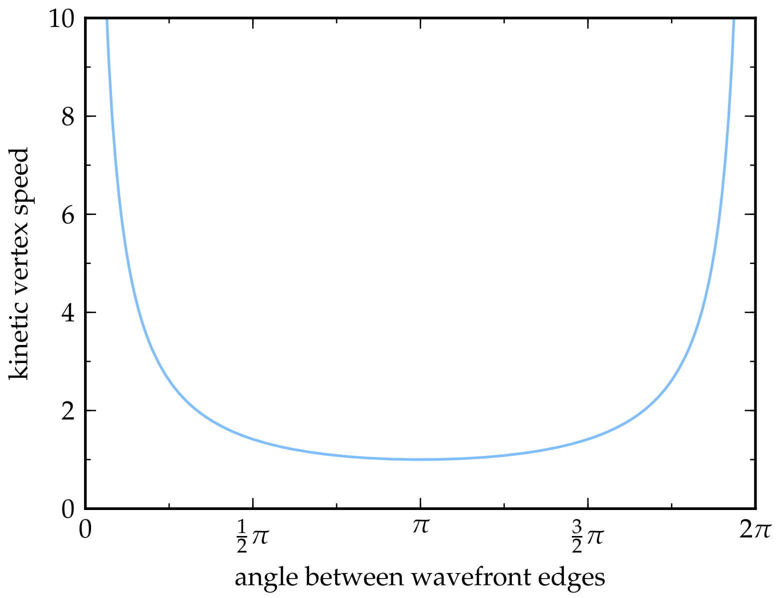


Figure 5: The speed of a kinetic vertex is a function of the angle between its defining wavefront edges.

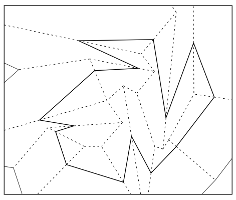
As shown in Figure 5, this function evaluates to 1 when α is π and goes towards infinity when the angle is either very small or almost a full circle. This matches our expectation that a vertex can move slowly when it moves orthogonal to the wavefront, in the same direction as the wavefront propagation, and has to move faster the closer it moves in a direction that is more parallel to its defining wavefront edges.

3.2 KINETIC TRIANGULATION

Once the initial wavefront has been constructed, a constrained triangulation T of the vertex set of G is computed. We require that all edges of G exist as edges of the triangulation. The triangulation edges that do not already exist in G are called *spokes*.

The kinetic triangulation $\mathcal{K}$ is created by adding the spokes of T to $\mathcal{W}(G)$. Care must be taken to attach spokes to the correct vertices of $\mathcal{W}(G)$ such that immediately following the start of the wavefront propagation process the part of the plane that now has been swept over by $\mathcal{W}(G)$ is void of any triangulation edges.

As previously stated, terminal vertices of G will cause edges with an initial length of zero to be added to the wavefront. When the propagation process starts, these edges will no longer be in their degenerated state but will instead have positive lengths. The faces incident to



these edges will now be quadrilaterals instead of triangles because T only triangulated the initial vertex set.

To achieve a partition into triangles only, additional spokes are added to the initial kinetic triangulation: for each terminal vertex v of G a spoke s is added at one of the vertices of $\mathcal{W}(G)$ that corresponds to v. This additional spoke is added in such a way that once the wavefront propagation has started, the face incident at the extra edge emanated from v is not a quadrilateral. Instead, the area which would have been covered by the quadrilateral is split into two triangles by s. Therefore, once the propagation process has started, the part of the plane which has not yet been visited by the wavefront is a correct triangulation as required.

The signed area of a triangle of $\mathcal{K}$ is a function of time. Since all wavefront vertices move at constant speed this function is a polynomial quadratic in time and therefore its roots, i. e., the points in time when the triangle collapses, can be computed using standard methods for solving quadratics. (Details are discussed in Section 4.3.)

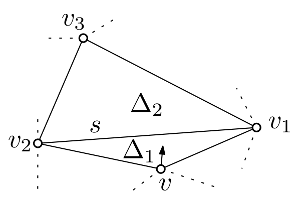
3.3 WAVEFRONT PROPAGATION PROCESS

The collapse times of all initial triangles are computed and triangles which collapse at a finite point in time are put into a priority queue Q. Whenever a triangle collapses, we have to update $\mathcal{K}$ accordingly. A triangle can collapse in one of three ways:

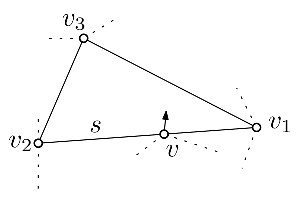
(a) A vertex moves over a spoke s. This type of event is called *flip event*: the spoke s is removed from $\mathcal{K}$ and the other diagonal of the remaining quadrilateral is added as a new spoke, as depicted in Figure 6. We remove the two original triangles from Q and add the two newly created triangles accordingly.

(b) A vertex v crashes into an opposing wavefront edge e. This *split event* is similar to the flip event mentioned previously as in both cases a triangle's vertex comes to lie on the vertex's opposing edge. It is also fundamentally different, however, since in a split event this vertex does not move over a spoke but instead crashes into the opposing edge, which is a wavefront edge. This crash causes the wavefront edge to be split into two parts.

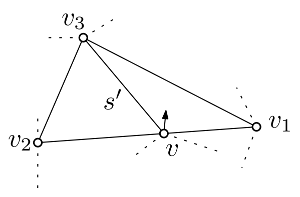
Two new vertices, v'_1 and v'_2, replace v, moving away from the crash location. The collapsed triangle Δ is removed from $\mathcal{K}$. Other triangles that were previously incident to v are updated to now be incident to either v'_1 or v'_2 and their collapse times in Q are updated. See Figure 7.



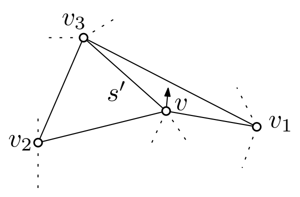
(a) Before a flip event.



(b) Δ_1 collapses.

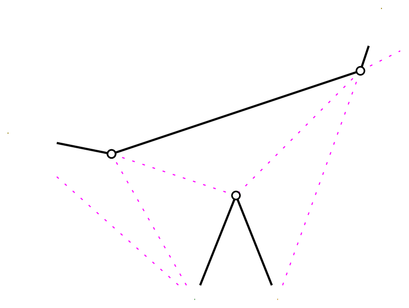


(c) After processing the flip event.

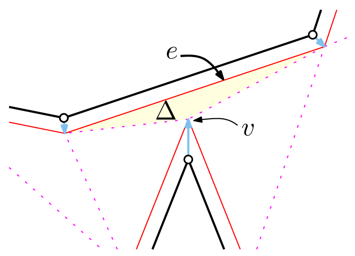


(d) Some time later.

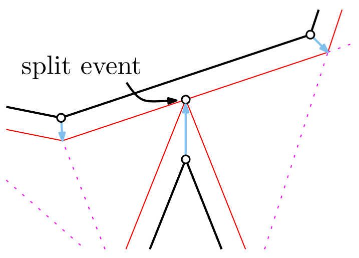
Figure 6: Flip event: Vertex v moves over the spoke s.



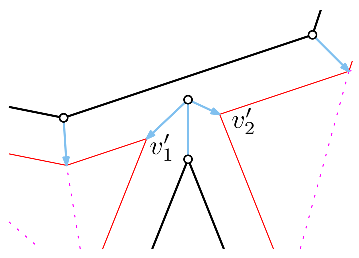
(a) Initial situation.



(b) Before the split event.

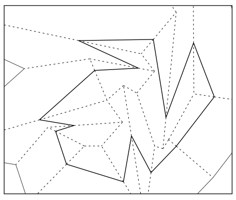


(c) Vertex v crashes into e.



(d) Some time later.

Figure 7: Split event: Vertex v crashes into wavefront edge e. Input edges are drawn heavy and in black, straight skeleton arcs and kinetic vertex traces in light blue, wavefront edges in red, and triangulation spokes are dotted in magenta. The wavefront propagation process is only shown on one side of the input.



(*c*) A triangulation edge collapses to zero length, causing two vertices, v_1 and v_2, to collide. If the collapsing edge was a wavefront edge this corresponds to an *edge event* as shown in Figure 8, otherwise it is a special case of a split event – one where two opposing vertices crash directly into one another, see Figure 9. Sometimes this is called a *multi split event* [Hub12].

In the case of an edge event, v_1 and v_2 merge into a new vertex v' with new direction and speed. In the case of a multi split event, v_1 and v_2 are replaced by vertices v'_1 and v'_2, moving away from the collapse point in different directions, just like in a normal split event. If one of the two new vertices is a reflex vertex again, the event is called a *vertex event* [EE99].

In either case the collapsed triangle is removed from Q and collapse times of all triangles now incident to v' or v'_1, v'_2 are updated in Q.

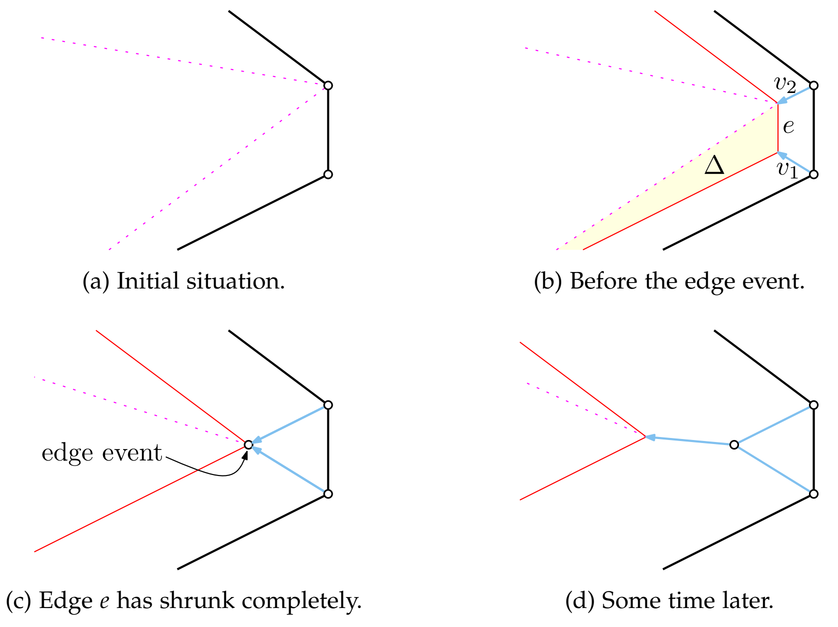
During the wavefront propagation process we have a concept of time. Time is monotonic, that is, it only moves forwards, never backwards. The propagation process starts at time zero.

When we compute collapse times, for instance during initial set up of the priority queue or during event handling when we update triangles, we are only interested in collapse times that happen either now or in the future. That is, we disregard any solution for a triangle's collapse time that is before the current time in the propagation process. If a triangle only has collapse times in the past, then we set its collapse time to infinity.

Once all triangles have collapsed and no more entries with finite collapse time remain in Q the propagation process ends. The nodes of the straight skeleton $\mathcal{S}$ are made up of the initial set of vertices of $\mathcal{K}$ at time zero and of all vertices that got stopped and replaced during an event, at their final position.

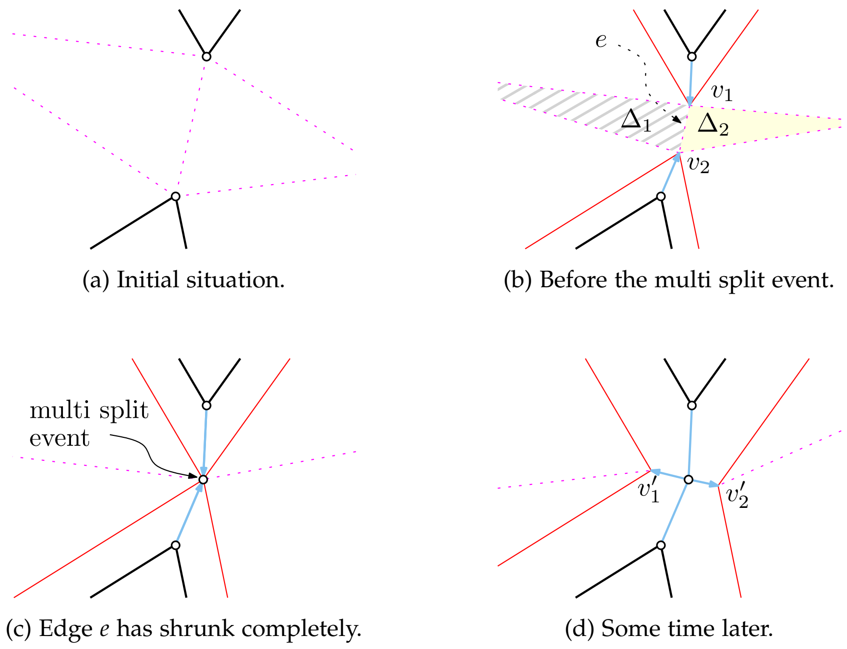
The arcs of the straight skeleton are the line segments traced out by kinetic vertices over their lifetime. Any kinetic vertices that remain in $\mathcal{K}$ at the end of the propagation process correspond to rays to infinity in $\mathcal{S}$.

We show the steps of the wavefront propagation, including kinetic triangulation, of an example in Appendix A, Figure 48.



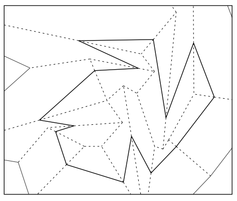
(a) Initial situation. (b) Before the edge event. (c) Edge e has shrunk completely. (d) Some time later.

Figure 8: Edge event: Wavefront vertex e shrinks to a length of zero as two neighboring vertices collide.



(a) Initial situation. (b) Before the multi split event. (c) Edge e has shrunk completely. (d) Some time later.

Figure 9: Multi-split event: Vertices v_1 and v_2 crash into each other as their connecting spoke shrinks to a length of zero and the two incident triangles collapse.



3.4 COMPLEXITY

3.4.1 *Space complexity*

Let G be an input graph with n vertices of which t are terminal vertices. The triangulation T will consist of $2 \cdot n - 2$ triangles. For creating $\mathcal{K}$ we add one extra triangle per terminal vertex, so $\mathcal{K}$ will initially consist of $2 \cdot n + t - 2$ triangles.

Edge and split events only remove triangles from $\mathcal{K}$; they never cause any triangles to be added. Flip events re-triangulate a quadrilateral, replacing two triangles with two different ones. Thus, the number of triangles is not increasing over time and at no point we will need to keep track of more than $2 \cdot n + t - 2 \in \mathcal{O}(n)$ triangles. This also limits the size of the priority queue which keeps track of collapse times. Furthermore, the $\mathcal{O}(n)$ bound trivially extends to the number of kinetic vertices we have to maintain at any time.

Since edge and split events reduce the number of triangles in $\mathcal{K}$ and nothing ever increases the size of $\mathcal{K}$, an $\mathcal{O}(n)$ upper bound on the number of edge and split events can be deduced. As mentioned before, the nodes of the straight skeleton are the union of the vertex set of $\mathcal{W}(G)$ at time zero and the set of nodes created during edge and split events. Therefore, the number of straight skeleton nodes is in $\mathcal{O}(n)$ and, since the straight skeleton is a planar graph [AA98], the size of the entire straight skeleton is linear in the input graph's size.

For an input graph G with n vertices the algorithm's overall space complexity is therefore in $\mathcal{O}(n)$.

3.4.2 *Run-time complexity.*

To analyze the run-time complexity of Aichholzer and Aurenhammer's triangulation-based algorithm, we have to look at its different stages.

SETUP. For an input graph G with n vertices the cost of constructing the initial wavefront and kinetic triangulation is dominated by the cost of actually triangulating the vertex set of G. Since G can be an arbitrary planar straight-line graph and not just a simple polygon, we know the run time complexity of triangulations to be in $\mathcal{O}(n \log n)$.

We need a priority queue to provide us with information on which event to handle next. The elements of this queue are the triangles of $\mathcal{K}$, and the priorities are the triangles' collapse times: the earlier the collapse, the sooner we have to handle the event.

We assume our priority queue has the following properties: Initial setup with n elements runs in $\mathcal{O}(n \log n)$. Removing any known element, not just the next one, can be done in $\mathcal{O}(\log n)$. Likewise, updating a known element to a new key costs $\mathcal{O}(\log n)$.

Computing the collapse times of all kinetic triangles is done in $\mathcal{O}(n)$ since we have $\mathcal{O}(n)$-many triangles and computing one collapse time can be done with a constant amount of work. Then constructing the initial priority queue costs $\Theta(n \log n)$.

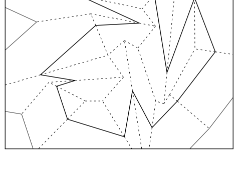
EDGE AND SPLIT EVENT HANDLING. For any event, de-queueing it from the priority queue is $\mathcal{O}(\log n)$ work, as discussed previously. However, for edge and split events this cost is dominated by the other damage such an event can cause: they both will replace vertices in up to $\mathcal{O}(n)$ triangles. This makes it necessary to re-compute the collapse times of all these triangles and to update their position in the priority queue. Therefore, the complexity of handling an edge or a split event is in $\mathcal{O}(n \log n)$.

We already established previously that the number of edge and split events is bound by $\mathcal{O}(n)$. Thus, the total cost of handling all non-flip events is $\mathcal{O}(n^2 \log n)$.

FLIP EVENT HANDLING. A flip event replaces two triangles with two different ones. Removing two triangles and adding two new triangles, or, equivalently, updating the collapse times of two triangles will cause only $\mathcal{O}(\log n)$ work per event. This leaves us to consider what the upper bound on the number of flip events is.

The number of kinetic vertices that exist during one run of the algorithm is in $\mathcal{O}(n)$: Let k_0 be the number of kinetic vertices that are created during the initial set up of the wavefront. It is easy to see that $k_0 \in \mathcal{O}(n)$: The input graph G is a planar graph with n vertices. Each of its $\mathcal{O}(n)$ edges contributes two edges to $\mathcal{W}(G)$ and each each of its $\mathcal{O}(n)$ terminal vertices contributes one edge to $\mathcal{W}(G)$. Therefore, the number of edges in $\mathcal{W}(G)$ is linear. All wavefront vertices have a degree of two, so their number is also linear, that is, $k_0 \in \mathcal{O}(n)$. Any edge or split event can add one or two new kinetic vertices. Since the number of edge and split events is bounded by $c \cdot n$ for a fixed constant c, the number of kinetic vertices added during the entire wavefront propagation process is bounded as well. This yields an upper bound of $\mathcal{O}(n)$ for the number of kinetic vertices.

Kinetic vertices move at constant speed along straight lines. The area spanned by three points in the plane moving at constant velocity is a quadratic polynomial in time. Therefore, three such points become collinear exactly once, twice, or never and so a kinetic triangle of three kinetic vertices will collapse at most twice.



Let k be the number of kinetic vertices. Then there can be at most $\binom{k}{3}$ different triangles since that is the number of possible combinations of three elements out of k.

From the fact that k is in $\mathcal{O}(n)$ and that each triangle collapses at most twice we can deduce that the number of flip events is in $\mathcal{O}(n^3)$.

Therefore, the total cost of handling all flip events is in $\mathcal{O}(n^3 \log n)$.

Note that this argument assumes that the input is in general position. For instance, three vertices can also be collinear throughout the entire propagation process because they move in the same direction and at the same speed and are collinear in their initial position. Even if the initial triangulation will not have a triangle of three such vertices — since a valid triangulation has no collapsed triangles — further events may cause such a triangle to exist multiple times. See Section 5.2 for a discussion on arbitrary input.

LOWER BOUNDS. We have just shown that handling all non-flip events requires at most $\mathcal{O}(n^2 \log n)$ work. For specific triangulations this bound is actually tight: Huber [Hub12, page 48] provides a convex polygon with a given triangulation that will result in $\Omega(n)$ edge events each affecting $\Omega(n)$ triangles (Figure 10). It is, however, not clear whether there exists an input that requires $\Omega(n^2 \log n)$ work for handling edge and split events for all possible triangulations.

The upper bound for the number of flip events is in $\mathcal{O}(n^3)$, as discussed previously. The best known lower bound is a linear factor below that: Huber and Held give a simple polygon in [HH10b] which causes $\Omega(n^2)$ many flip events for all valid triangulations (Figure 11). Even re-triangulating during the propagation process would not improve the run-time behavior. Finding tighter bounds is an open problem.

Note that $\mathcal{O}(n^3)$ is a worst-case bound. In practice, for real-world input, we have seen far more docile numbers for flip events. See Chapter 6.

SUMMARY. The potential cost of flip-event handling dominates all other parts of the algorithm. Hence the overall upper bound for the run-time complexity of the triangulation-based algorithm is in $\mathcal{O}(n^3 \log n)$. This is the best known upper bound.

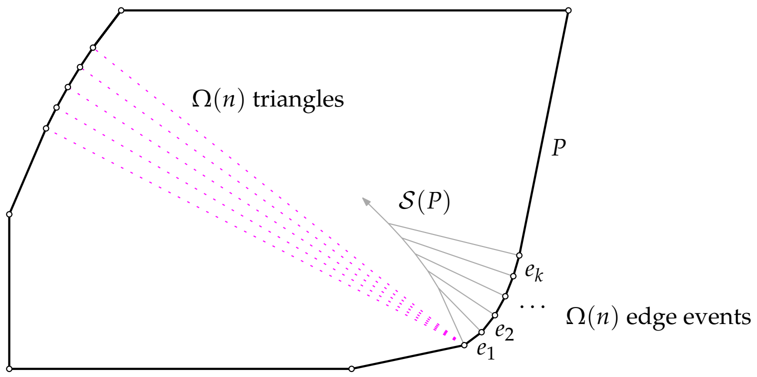


Figure 10: Even a convex polygon can have $\Omega(n)$ many edge events that each affect $\Omega(n)$ triangles. (Based on figure from [Hub12].)

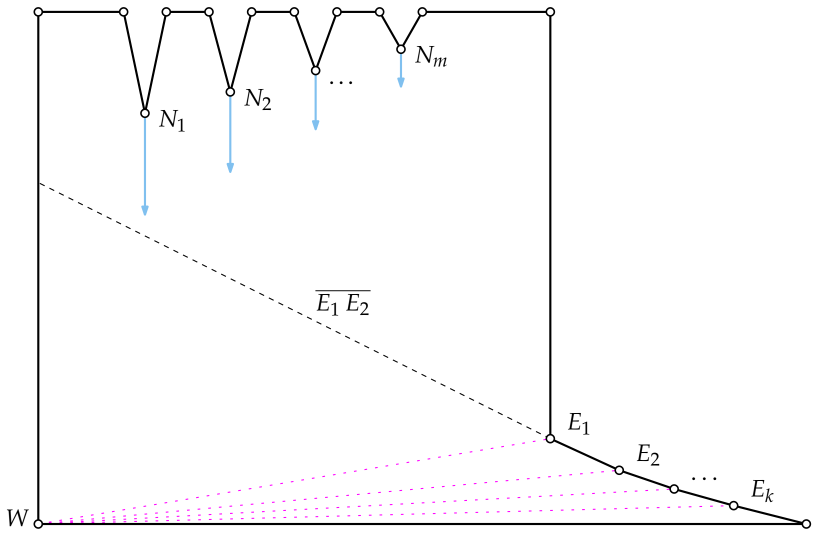
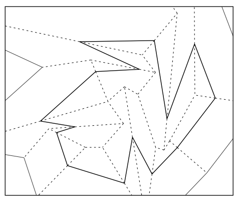


Figure 11: This particular input causes a total of $\Omega(n^2)$ flip events. By construction no triangulation is possible that results in fewer flips. (Based on figure from [HH10b].)



4 IMPLEMENTATION

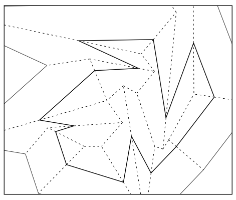
We implemented the triangulation-based algorithm outlined in the previous chapter as a library. This library, named Surfer, is written in the C programming language. We created two frontends that make use of it: The first is a command line tool called `surf` which loads an input graph and writes out the straight skeleton. The second one, `surfgl`, is a GUI program which, given an input graph, can step through the individual events of the wavefront propagation process. At each step it shows the current state of the algorithm, including current wavefront, kinetic vertices and triangulation, straight skeleton nodes and arcs, and the next collapsing triangle. It uses OpenGL [OGL] rendering and allows zooming to investigate areas of particular interest.

Output formats supported by `surf` are (i) the XML based format used by the Ipe [Che] drawing editor, (ii) the format used by GeoGebra [Hoh], an interactive geometry application, and (iii) PNG, the portable network graphics format. The PNG output is the only non-vector format: we raster the straight skeleton and the input graph onto a 8192 pixels by 8192 pixels large image. Its main purpose is to be able to quickly compare the skeletons produced by `surf` with those produced by other implementations like Huber's Bone [HH11], which we modified to support the same output format. Furthermore, the tool can be run in a mode where it will not write any output at all. That feature is useful when one only cares about timing the computation of the straight skeleton itself.

4.1 DATA STRUCTURES

In order to represent the kinetic triangulation, our implementation keeps a list of kinetic vertices and a list of kinetic triangles. Furthermore, we maintain a list of straight skeleton nodes created so far.

Vertices move with time, at linear speed. So we keep for each vertex its *velocity* and an *origin*, which is the position in the plane where it was at time zero – or would have been if it had already existed at that time. We also store the direction of its incident wavefront edges which is required to compute the direction of a new wavefront vertex during event handling. Additionally we store two timestamps, one for the time at which the vertex was created and one where it



stopped. Kinetic vertices start and stop in skeleton nodes. Therefore we include a reference to the respective skeleton nodes in the vertex structure as well.

A straight skeleton node is nothing more but a set of coordinates in the plane and any context about its setting in the actual straight skeleton graph, such as which arcs are incident to it, can be derived from the information in the kinetic vertex list.

```
struct sknode_t {
  vector_t pos;
};

struct kvertex_t {
  vector_t o, v, ccw_wavefront, cw_wavefront;
  real_t starts_at, stops_at;
  sknode_t* start_node, stop_node;
};
```

For each triangle we store three pointers to its incident vertices and three pointers to its neighbor triangles. A neighbor pointer that is NULL indicates a wavefront edge on this side of the triangle.

```
struct ktriangle_t {
  kvertex_t *v[3];
  struct ktriangle_t *n[3];
  int heap_position;
};
```

Elements in a triangle's vertex and neighborlist are ordered counterclockwise and in such a manner that `v[0]` is opposite of `n[0]`. Additionally we store a reference to this triangle's location in the heap that represents our priority queue. This is useful when we need to find, replace, or update a triangle that is not at the top of the heap. This reference requires updating whenever the priority queue re-orders elements.

4.2 LOADING AND TRIANGULATING THE INPUT

Our code supports loading of input graphs in several different formats customarily used in computational geometry. This includes the format used by the Ipe [Che] drawing editor, the various formats used by Martin Held's large set of test data and the format understood by Shewchuk's excellent triangulation code, `triangle` [She96]. For some of these formats we use parsing code generously provided by Stefan Huber.

After loading the input, our library scales and translates the graph so that all vertices lie in a square of edge length two centered at the origin.

We then use `triangle` to construct a constrained triangulation of the PSLG. However, the Surfer library has been written with modularity in mind, so replacing `triangle` with a different implementation is possible.

Shewchuk's library already directly computes a triangle list with triangle adjacency links, so creating the set of kinetic triangles from that is straight forward. Some additional processing is needed, however, because `triangle` constructs a triangulation of the input's convex hull only. We need to add the extra triangles to attain a triangulation of the entire plane ourselves. See Section 4.6 later in this chapter.

As mentioned in Section 4.1 previously, we store the wavefront implicitly in the set of triangles: If a triangle's neighbor pointer on an edge is NULL, then this edge is a wavefront edge. The adjacency relations that the triangulation code produces will still list neighbors across input edges. In order to create the initial wavefront, we clear the neighborhood relation at all triangle edges that correspond to PSLG edges.

Next, we have to add extra triangles for terminal input vertices as already described in Section 3.2.

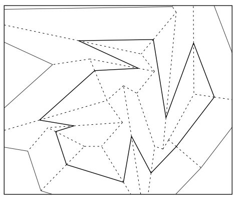
Once we constructed the initial kinetic triangulation, we create all kinetic vertices, computing their speed as outlined in Section 3.1.

As the last step before we can start the wavefront propagation process, we need to initialize the priority queue. We add all kinetic triangles with their corresponding collapse times.

4.3 COLLAPSE TIMES

During the wavefront propagation process we constantly have to update the kinetic triangulation $\mathcal{K}$ in order to guarantee its consistency. The points in time when we have to update $\mathcal{K}$ correspond to triangles collapsing to either straight-line segments or to just points in the plane. It is important that these events are handled in the correct order since an event at time t can (i) cause an event previously scheduled to happen at time $t' \geq t$ to no longer happen and (ii) can create new events at times $t' \geq t$.

To process events in their correct order, we maintain a priority queue, a min-heap, with collapse times of all triangles in $\mathcal{K}$ as its key.



In this section we describe how to compute the collapse time of a triangle in different ways.

4.3.1 *Collapsing area*

For a given triangle Δ of our kinetic triangulation $\mathcal{K}$ we know that its vertices v_1, v_2, v_3 move at constant speed and direction. We can thus parameterize the position of v_i at time t as

$$v_i(t) = o_i + t \cdot s_i \ ,$$

where o_i is the vertex's position at time $t = 0$ and s_i is its velocity vector. By $v_{i,x}$ and $v_{i,y}$ we refer to the x respectively y coordinates of v_i. Similarly, with $o_{i,x}$, $o_{i,y}$, $s_{i,x}$, and $s_{i,y}$ we mean the coordinates of the position and velocity vectors.

Note that not all kinetic vertices will already exist at time zero. We nevertheless define v_i this way. For a vertex that is created at a later time we need to compute its position o, that is, the position where it would have been at $t = 0$, based on where it is at the time of its creation and its velocity.

As already summarized in Chapter 3, in the general case in order to compute the next event caused by Δ, we have to consider the area $A_\Delta(t)$ of Δ over time t. The signed area can be computed using the determinant method for triangles in the plane:

$$A_\Delta(t) = \frac{1}{2} \cdot \begin{vmatrix} v_{1,x} & v_{1,y} & 1 \\ v_{2,x} & v_{2,y} & 1 \\ v_{3,x} & v_{3,y} & 1 \end{vmatrix} .$$

Expanding the determinant yields

$$\begin{aligned} A_\Delta(t) = \frac{1}{2} \cdot (\quad & (v_{1,x}\, v_{2,y} + v_{2,x}\, v_{3,y} + v_{3,x}\, v_{1,y}) + \\ & -(v_{1,y}\, v_{2,x} + v_{2,y}\, v_{3,x} + v_{3,y}\, v_{1,x}) \quad). \end{aligned}$$

Note that in this term $v_{i,x}$ and $v_{i,y}$ are still functions of time. Using the definition of $v_i(t)$ provided above, this expands into the somewhat ungainly quadratic in t:

$$\begin{aligned}
A_\Delta(t) = \tfrac{1}{2} \cdot (\quad t^2 \cdot (\quad & s_{1,y}\, s_{2,x} + s_{1,x}\, s_{2,y} + s_{1,y}\, s_{3,x} \\
& - s_{2,y}\, s_{3,x} - s_{1,x}\, s_{3,y} + s_{2,x}\, s_{3,y} \quad) + \\
t \cdot (\quad & o_{2,y}\, s_{1,x} - o_{3,y}\, s_{1,x} - o_{2,x}\, s_{1,y} \\
& + o_{3,x}\, s_{1,y} - o_{1,y}\, s_{2,x} + o_{3,y}\, s_{2,x} \\
& + o_{1,x}\, s_{2,y} - o_{3,x}\, s_{2,y} + o_{1,y}\, s_{3,x} \\
& - o_{2,y}\, s_{3,x} - o_{1,x}\, s_{3,y} + o_{2,x}\, s_{3,y} \quad) + \\
(\quad & - o_{1,y}\, o_{2,x} + o_{1,x}\, o_{2,y} + o_{1,y}\, o_{3,x} \\
& - o_{2,y}\, o_{3,x} - o_{1,x}\, o_{3,y} + o_{2,x}\, o_{3,y} \quad) \quad).
\end{aligned}$$

The roots of this function are the times when the triangle Δ collapses and causes events.

4.3.2 *Collapsing edges*

Another approach to consider is looking at edge collapse times: whenever a triangle's edge shrinks to a length of zero, then clearly the triangle's area also becomes zero.

Consider two kinetic vertices, v_1 and v_2, and the line segment between them as a vector $\overrightarrow{e} = \overrightarrow{v_1 v_2} = (o_2 - o_1) + t \cdot (s_2 - s_1)$. The square of its length is

$$d^2(t) = \overrightarrow{e} \cdot \overrightarrow{e}.$$

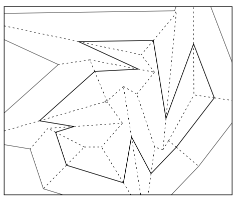
The zero of the derivative of d^2 gives us the time of closest approach as a simple fraction of two dot products:

$$t_e = \frac{(s_1 - s_2)(o_2 - o_1)}{(s_1 - s_2)(s_1 - s_2)}.$$

If v_1 and v_2 are the endpoints of an edge e and e is a wavefront edge, we know that they will eventually coincide, have coincided in the past[1], or are running in parallel. Therefore, if the denominator of the fraction is not zero, we can easily compute the collapse time t_e of e. If e is not a wavefront edge, then it is not guaranteed that it will ever collapse. As such, we have to evaluate d^2 at time t_e. If a comparison to zero yields true, then the edge will collapse at time t_e. Otherwise it will never collapse.

Obviously, not all types of triangle collapses are witnessed by collapsing edges. As such, this method cannot help us to compute the next collapse time of a given triangle in all cases.

1 Note that *in the past* also includes times before time zero, that is, before the wavefront propagation even started.



4.3.3 *Vertices crashing into edges*

In triangles that have at least one wavefront edge we can leverage the fact that wavefronts propagate at unit speed in a self-parallel manner to compute the time when the opposing vertex v crashes to the wavefront edge e or crosses over its supporting line. See Figure 12.

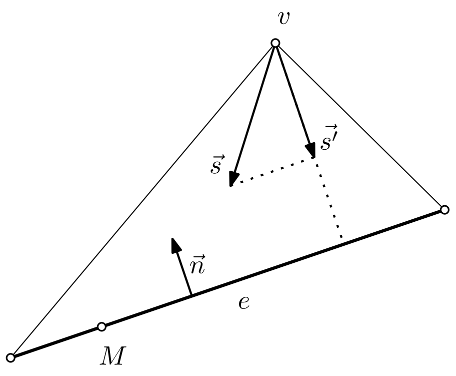


Figure 12: Vertex v moving towards wavefront edge e.

Let $\overrightarrow{s}$ be the velocity of v and let $\overrightarrow{n}$ be a unit normal of e in the direction of its propagation. Let $\overrightarrow{s'}$ be a projection of $\overrightarrow{s}$ onto $\overrightarrow{n}$. Then $\overrightarrow{s'}$ approaches e at a speed of

$$\left|\overrightarrow{s'}\right| = \overrightarrow{s} \cdot (-\overrightarrow{n}).$$

While v approaches the wavefront edge e, so, likewise, e moves towards v. Wavefront edges move at unit speed in a self parallel manner, thus the combined speed of approach is $1 - \overrightarrow{s} \cdot \overrightarrow{n}$.

The distance between v and e is $\overrightarrow{Mv} \cdot \overrightarrow{n}$ where M is an arbitrary point on the supporting line of e and $\overrightarrow{Mv}$ is the vector from M to v.

Putting it all together, we can compute the time t_v of when v lies on the supporting line of e as

$$t_v = \frac{\overrightarrow{Mv} \cdot \overrightarrow{n}}{1 - \overrightarrow{s} \cdot \overrightarrow{n}},$$

where v is the position of the kinetic vertex at time zero and M is a point on the supporting line of e at time zero.

4.3.4 *Motivation for using different methods*

Computing the roots of the quadratic function from Section 4.3.1 can become quite inaccurate depending on its numerical conditioning. Consider the equilateral triangle Δ in Figure 13. It is bounded by three wavefront edges and thus will collapse when all three of its vertices become coincident in the center of Δ.

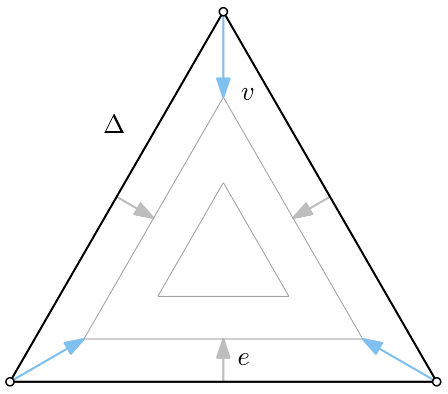


Figure 13: A collapsing triangle.

We plotted the area of Δ as a function of time as the blue graph in Figure 14. When the area becomes zero at time $5/\sqrt{3}$, the triangle collapses and causes a change in the wavefront topology. Obviously, even slight numerical inaccuracies might cause our function to be different. Such a difference could result in the quadratic function having no real roots. Therefore, we would miss this event entirely.

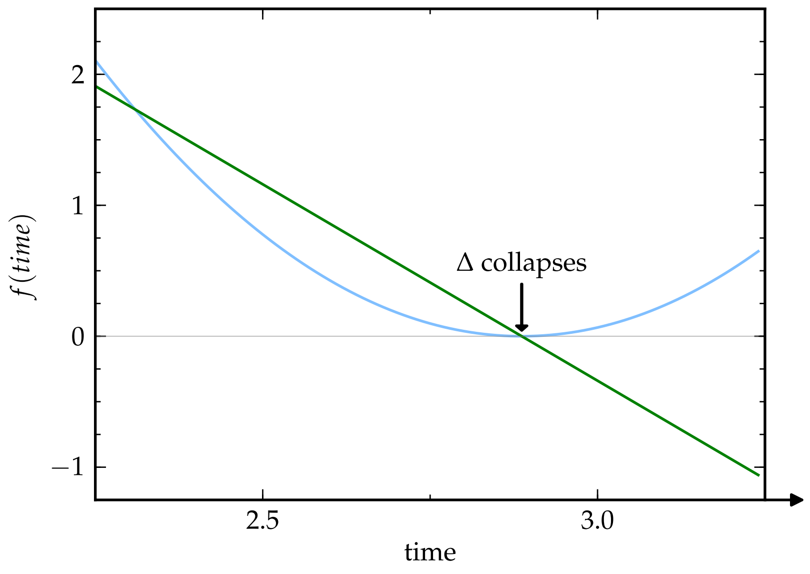
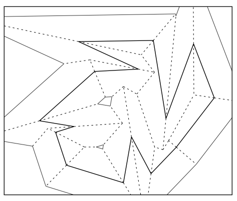


Figure 14: Computing collapse times of a triangle using two different strategies. The blue parabola represents the signed area of the triangle from Figure 13 over time, the green line shows the distance from the kinetic vertex v to the wavefront edge e.



A plot of the distance from the kinetic vertex v to its opposite wavefront edge e is shown in green in Figure 14. This is the approach for finding collapse times described in Section 4.3.3. Since v is approaching e, this linear function will evaluate to zero at one point in time. Slight errors in input will only result in slight errors for the collapse time, not in missing the collapse entirely.

Avoiding the area method for finding collapse times therefore results in a more robust implementation of the triangulation-based algorithm.

4.4 EVENT CLASSIFICATION

In the previous section we outlined different approaches to compute the collapse times of triangles. We have, however, not yet described when to use which approach.

Keep in mind that the purpose of determining the collapse time of a triangle is to process the change in the kinetic triangulation and possibly the wavefront itself at this particular time.

Therefore, when populating the priority queue, we try not only to find the collapse time for a given triangle but also, if possible, to determine the type of event this will be. For this purpose we can use additional information we can learn from the kinetic triangulation $\mathcal{K}$, such as which edges of a triangle are wavefront edges and which ones are spokes.

We can distinguish the following cases based on how many edges of a triangle are wavefront edges:

(*a*) If a triangle Δ has three wavefront edges, then the area it covers essentially is a simple polygon. This polygon has been split off from the rest of the as yet unvisited plane and is completely independent of any outside events. As the wavefront propagates, the polygon's area shrinks until it collapses entirely. At this particular time all three wavefront edges collapse to zero length simultaneously.

For triangles in this category we compute collapse times using the edge collapse time method described previously.

(*b*) A triangle with exactly two wavefront edges can collapse in two distinct ways. Either exactly one of the two wavefront edges collapses to zero length or all three of its sides collapse at the same time.

To find the collapse times of these triangles, we use the earlier of the two edge collapse times of the wavefront edges, ignoring any times in the past.

(*c*) Collapses for triangles with exactly one wavefront edge e appear in one of two forms:

- The wavefront edge can collapse, causing a classic edge event.
- Consider the vertex v which lies opposite the wavefront edge e. This vertex can crash into e or sweep across its supporting line.

In order to determine which of these two cases happens, we compute both the edge collapse time t_e of e, as well as the vertex crash time t_v of v, using two of the procedures previously discussed. As always, we ignore collapse times in the past.

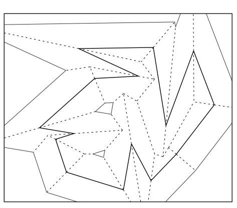
If t_e is earlier than t_v or if t_e equals t_v, then this event is an edge event, as e collapses at that time. If t_v happens strictly earlier than t_e, then this event is either a split event or a flip event. In order to classify this event, we compute the length of all sides of the triangle at time t_v. If the longest edge is the wavefront edge, it is a split event, otherwise it is a flip event.

(*d*) A triangle that is bounded only by spokes can either collapse due to a flip event, that is, a vertex can sweep across its opposing spoke, or because one of its spokes collapses to zero length.

For each edge of a triangle we compute its collapse time, if it exists. We also compute the time when the triangle's area becomes zero using the determinant approach.

If the time obtained from the determinant approach is earlier than any edge collapses this triangle will cause a flip event at that time.

In the other case two opposing vertices will crash into each other as a spoke collapses. Some authors, such as Huber in [Hub12], define this to be a split event because it involves at least one reflex wavefront vertex. For our purposes we will still call this an edge event since its handling is identical to the case where the vanishing spoke was indeed an edge of the wavefront.



4.5 EVENT HANDLING

4.5.1 *Handling Edge Events*

Our procedure to handle edge events can deal with true edge events, where a wavefront edge collapses to zero length, as well as with false edge events, sometimes called multi split events, where a spoke collapses as two opposing wavefront vertices crash into one another. Figure 15 sketeches the situation before a true edge event.

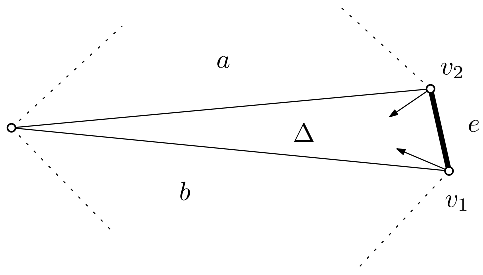


Figure 15: Edge event: Edge e is about to collapse.

The algorithm's purpose is to stop the two wavefront vertices that have now become incident as their connecting edge collapsed, to create a new kinetic vertex, and to remove the collapsed triangle Δ from the triangulation. Furthermore, it needs to maintain the consistency of our kinetic triangulation.

If the collapsing edge was not a wavefront edge we also schedule the handling of the incident neighbor to happen immediately.

Algorithm 1 briefly sketches our event handling code. The collapsing edge is passed to the function as an index in the given triangle's neighborlist. First, it fetches references to the two colliding vertices v_1 and v_2 and then marks them as stopped using the `stop_kvertices()` function. If necessary, this function also creates a new straight skeleton node, *sk_node*, at the position of these vertices and adds it to the global node list. It amends information stored with v_1 and v_2 to record the node at which they have stopped. Then a new kinetic vertex v is created using `compute_new_kvertex()`. That function also tags the new vertex as having started in *sk_node*.

In the second part of the procedure we have to update the neighborhood relations of the vanishing triangle's neighbors: in the neighborhood list of a we replace Δ with b; in b we replace Δ with a.

The helper function `find_index_of_t(neighbor, triangle)` is used to find the position of *neighbor* in *triangles*'s neighborlist.

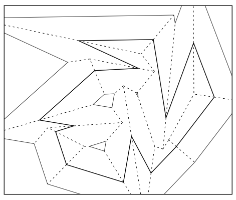
Algorithm 1 Edge Event Handling

```
 1: procedure HANDLE_EDGE_EVENT($t, e, now$)
 2:     v1 ← t->v[(e+1)%3]                              ▷ Find incident vertices
 3:     v2 ← t->v[(e+2)%3]
 4:     sk_node ← stop_kvertices(v1, v2, now)
 5:     v ← compute_new_kvertex(
            v1->ccw_wavefront, v2->cw_wavefront, now, sk_node)

                                              ▷ Update neighborhood relationships
 6:     a ← t->n[(e+1)%3]                               ▷ Find neighboring triangles
 7:     b ← t->n[(e+2)%3]
 8:     if $a \neq$ NULL then
 9:         a->n[ find_index_of_t(a, t) ] ← $b$
10:         replace_kvertex_ccw(a, v2, v)
11:     end if
12:     if $b \neq$ NULL then
13:         b->[ find_index_of_t(b, t) ] ← $a$
14:         replace_kvertex_cw(b, v1, v)
15:     end if

16:     n ← t->n[e]                                     ▷ Handle wavefront edge
17:     if n $\neq$ NULL then
18:         n->n[ find_index_of_t(n, t) ] ← NULL
19:         schedule_immediately(n)
20:     end if
21: end procedure
```

The replace_kvertex_*() procedures replace all references to the old vertices v_1 or v_2 with references to the newly created vertex v, iterating around the pivot vertex either clockwise or counter-clockwise. For any affected triangle it needs to re-compute its collapse time and update the priority queue accordingly.

Lastly, if the vanishing edge was not a wavefront edge we inform our event scheduling code that it should handle the collapse of the neighbor on the edge e next. It too will have an edge event at the same time and processing immediately ensures that we do not run into any issues with globally inconsistent triangulations.

Ignoring potential memory management costs when creating a new straight skeleton node, all operations but one in this algorithm run in constant time. The one exception is replace_kvertex_*(): If n is the number of triangles in $\mathcal{K}$, then the number of triangles potentially sharing v_1 or v_2 is in $\mathcal{O}(n)$. The cost of updating a vertex in a single triangle is constant and updating its collapse time in the priority queue is logarithmic. Therefore the time complexity of an update of all triangles affected by a single edge event combined is in $\mathcal{O}(n \log n)$.



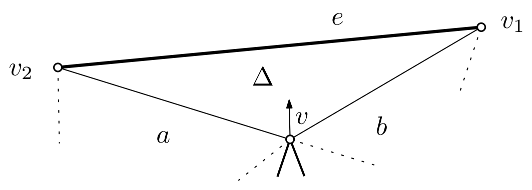
Since this dominates all other cost, the time complexity of handling a single edge event also is in $\mathcal{O}(n \log n)$.

4.5.2 *Handling Split Events*

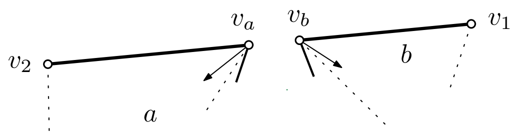
In a split event a vertex v crashes into its opposing wavefront edge. We have outlined the procedure to handle such an event in Algorithm 2. The vertex v is stopped at this point, a new skeleton node is produced, and two new vertices, v_a and v_b, are instantiated.

Consider the area that is not yet covered by the wavefront and that the collapsing triangle was part of. If, prior to the split event, this area was simply connected, then the split event has cut it into two disconnected regions. One of the two new vertices v_a and v_b is in one of the disconnected regions, the other vertex is in the other.

As before, we have to make sure that the kinetic triangulation remains valid. Let Δ be the triangle that collapsed and let a and b be neighbors of Δ as depicted in Figure 16. Then in both a and b the spoke that was previously shared with Δ now becomes a wavefront edge. These two new wavefront edges partition e and replace it in the wavefront set.



(a)



(b)

Figure 16: Split event: (a) immediately before vertex v crashes into e; (b) the situation after the split.

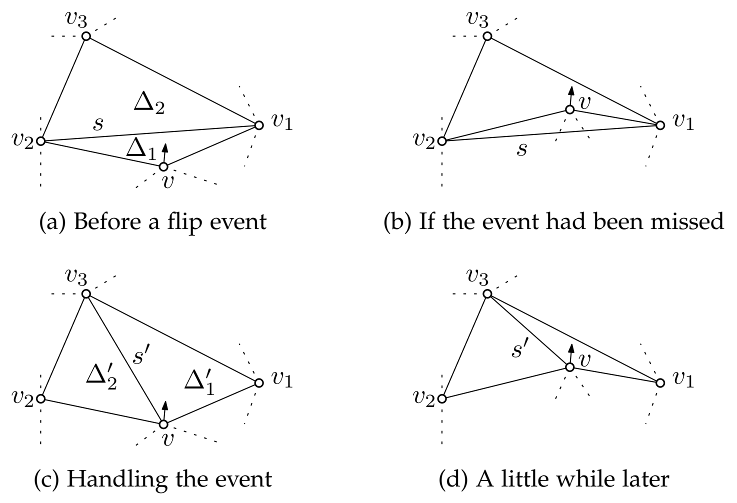
Furthermore, all triangles incident to v have to be updated to now be incident to one of v_a or v_b, depending on which side of the divide they are.

The time complexity for handling a single split event is identical to the edge event described above and for the same reason: Marking v stopped, creating a new skeleton node and adding two new kinetic vertices is dominated by the potential cost of touching a linear number of triangles, those previously incident to v, each requiring an update in the priority queue for a total of $\mathcal{O}(n \log n)$ work.

Algorithm 2 Split Event Handling

```
 1: procedure HANDLE_SPLIT_EVENT(t, e, now)
 2:     v ← t->v[e]
 3:     v1 ← t->v[(e+1)%3]
 4:     v2 ← t->v[(e+2)%3]
 5:     sk_node ← stop_kvertex(v, now)
 6:     va ← compute_new_kvertex(
            v->ccw_wavefront, v2->ccw_wavefront, now, sk_node)
 7:     vb ← compute_new_kvertex(
            v1->cw_wavefront, v->cw_wavefront, now, sk_node)

 8:     a ← t->n[(e+1)%3]
 9:     a->n[ find_index_of_t(a, t) ] ← NULL
10:     replace_kvertex_ccw(a, v, va)

11:     b ← t->n[(e+2)%3]
12:     b->[ find_index_of_t(b, t) ] ← NULL
13:     replace_kvertex_cw(b, v, vb)
14: end procedure
```

4.5.3 *Handling Flip Events*

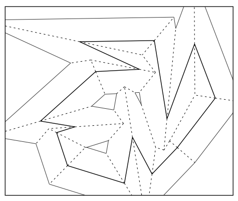


(a) Before a flip event (b) If the event had been missed

(c) Handling the event (d) A little while later

Figure 17: Flip event: Vertex v moves over the spoke s.

A flip event is the only type of event that does not coincide with a topological change of the wavefront. During the handling of a flip event no new straight skeleton arcs or nodes are created.



The purpose of the flip-event handler is to ensure our kinetic triangulation remains a valid triangulation of the part of the plane not yet visited by the wavefront.

Consider triangle Δ_1 in Figure 17a. As the vertex v moves towards the spoke s, this triangle will collapse. If we did not do any special processing and v moves a little bit further we would end up in a situation like in Figure 17b. This is not a valid triangulation anymore since clearly it does not partition the plane — there are parts that are covered by both $\Delta(v, v_1, v_2)$ and $\Delta(v_1, v_3, v_2)$.

To maintain a correct triangulation, we need to remove the spoke s and replace it with the other possible spoke s' in what is now a quadrilateral.

In our implementation that boils down to removing both triangles Δ_1 and its neighbor Δ_2 from the set of triangles and the priority queue. Then we add new triangles Δ_1' and Δ_2' in their place. We also need to update the local neighborhood information in the adjoining triangles.

When it comes to time complexity, a flip event is cheap compared to the two other event types. The dominating cost comes from having to remove both Δ_1 and Δ_2 from the priority queue and then adding Δ_1' and Δ_2' to it. The triangle Δ_1 is at the top of the heap as it is collapsing and the triangle whose event we are currently handling. The other triangle we need to remove, Δ_2 is not at the top. We use the heap position reference that we store with each triangle to find it in the heap structure (Section 4.1). Each removal and addition operation can be done at $\mathcal{O}(\log n)$ cost. Actually creating Δ_1' and Δ_2' and determining their neighborhood relations and updating their neighbors accordingly has constant cost. Therefore, a flip event requires work of total complexity in $\mathcal{O}(\log n)$.

4.6 UNBOUNDED TRIANGLES

The correctness of the algorithm is based on the fact that each change in the wavefront's topology is witnessed by one collapsing triangle of a kinetic triangulation of the plane. Not much consideration is given in the original paper on what exactly is meant by the phrase *triangulation of the plane*. We will show that a triangulation of just the convex hull of the input vertex set is not in itself sufficient. We will then outline possible approaches, including the one we implemented.

TRIANGULATING THE CONVEX HULL. Consider an input graph G like the one partially shown in Figure 18a. Input edges are bold while triangulation spokes are dotted.

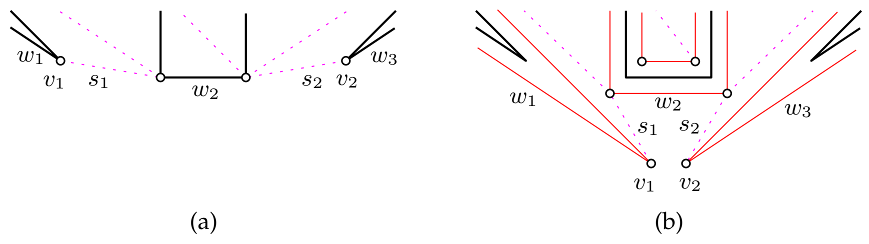


Figure 18: Subfigure (a) shows part of an input graph and an initial kinetic triangulation. In Subfigure (b) we see the wavefront some time after the propagation process has started. Observe that soon vertices v_1 and v_2 will collide and thus the wavefront topology will change. However, no triangle will collapse to observe that event!

The wavefront edges w_1, w_2 and w_3 have been emanated from input edges that are on $\mathcal{CH}(G)$, the convex hull of the input. Therefore, these wavefront edges are also on $\mathcal{CH}(\mathcal{K}_0)$, the convex hull of the kinetic triangulation at time zero.

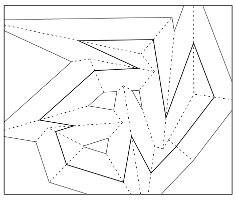
If we now start the wavefront propagation process and move forward in time, the situation will soon look like in Figure 18b. So far no triangle has collapsed, so no event handling was required. Vertices v_1 and v_2 are about to collide. However, no triangle exists that will collapse at the time of the collission to witness the corresponding change of the wavefront's topology. This contradicts the main principle for the correctness of this algorithm.

The problem is that a triangulation of the convex hull of kinetic vertices at time zero does not stay a triangulation of the convex hull once vertices move. Our existing rules governing updating the triangulation do not keep track of changes to the convex hull and, thus, the algorithm ends up with areas within $\mathcal{CH}$ that are not covered by the triangulation. This causes us to miss changes in this region.

One possible solution is to not just triangulate the convex hull of the initial input and then run the propagation process, but to maintain a full triangulation of the convex hull of the wavefront at all times. In that case in our example v_1 and v_2, being neighbors on $\mathcal{CH}(\mathcal{K}_t)$, would have shared a triangulation spoke and thus there would have existed a triangle to witness their collapse.

Two kinds of changes affect $\mathcal{CH}(\mathcal{K}_t)$:

- Vertices are added to it as a wavefront vertex v moves over a triangulation spoke e on the convex hull. We would already notice this in our existing framework as it corresponds to the triangle that e shares with v collapsing in what is essentially a flip event. Note that there would not exist a triangle to flip into, but that should not be a problem as that triangle would have been removed anyway during the flip-event handling.



- Vertices can move towards the interior and thus will no longer be on the convex hull. This is what happened in our example in Figure 18. This will happen whenever three consecutive vertices on $\mathcal{CH}(\mathcal{K}_t)$ become collinear. A new triangle would have to be added to $\mathcal{K}_t$ at that point, consisting of the three vertices that became collinear. The vertex in the middle would be removed from $\mathcal{CH}(\mathcal{K}_t)$.

We chose not to use this approach for two reasons. First, updating the convex hull when vertices leave the set requires having a second source of event times. Second, having vertices leave $\mathcal{CH}(\mathcal{K}_t)$ requires adding more triangles during the wavefront propagation process. We prefer having triangles only be removed and never added as this makes maintaining our data structures for the triangulation and priority queue easier and more efficient.

TRIANGULATING THE PLANE. Another approach is to triangulate a portion of the plane that is "big enough", that is, one that would cover all parts of the plane that will ever see an event and thus contain a straight skeleton node.

Unfortunately that seems not promising: One way to do that would be to select an area that is just big enough. This idea fails due to the fact that there are no known good heuristics to tell us how far out straight skeleton nodes will lie. We therefore have no means to know how much area on top of the input graph our triangulation needs to cover.

Another strategy might be to pick an area that is truly huge so that all nodes are almost guaranteed to be in this region. However, this will result in lots of very thin triangles. Such triangles are hard to manage on systems using finite-precision floating-point operations.

We have chosen the following approach in our implementation: First, create a constrained triangulation of the convex hull of the input graph. Then, for every edge e on the convex hull add one *unbounded triangle*, that is, one triangle with e as one edge and two infinite edges going outwards. All these edges are thought to meet at infinity.

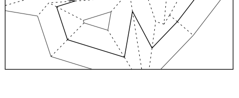
This raises the new question not addressed by Aichholzer and Aurenhammer of when such unbounded triangles collapse. It is insufficient to consider an unbounded triangle to collapse only when its finite edge is collapsing to zero length as this would allow unbound triangles to move to the interior of the wavefront. For an example where this happens refer to Figure 18 once more. During the initial triangulation process we would have created an infinite triangle for the convex hull edge w_2. The corresponding triangle would not collapse to witness the crash of v_1 into v_2, nor would any other.

Our solution is to consider a projection of the plane $\mathbb{R}^2$ to the sphere S^2. We choose the inverse of a stereographical projection from the north pole onto the plane through the equator. In particular, this projection then maps the origin to the south pole of S^2. The north pole represents all points at infinity. Every triangle of the triangulation in $\mathbb{R}^2$, unbounded or not, maps to one spherical triangle on S^2 whose edges lie on great circles. Unbounded edges of unbounded triangles map to finite edges lying on a great circle through the north and south pole. The triangulation of the plane that we have created maps to a proper triangulation of the sphere.

This projection does not preserve areas or distances. It does, however, preserve angles between lines. We do not make use of this property directly, but we do make use of one of its implications, namely that the collapse of a spherical triangle on S^2 implies its counterpart in $\mathbb{R}^2$ has collapsed also.

We now consider an unbounded triangle in $\mathbb{R}^2$ to collapse whenever its corresponding triangle on S^2 collapses, that is, when its three vertices lie on a common great circle. The events that can happen whenever such a triangle collapses are: (i) edge events where the finite edges collapses, or (ii) flip events. Both types of events can be handled in a very similar manner to how these events are dealt with when they involve bounded triangles only. Unbounded triangles can never be involved in split events.

Naturally, it would be cumbersome to actually have to compute a projection every time we wanted to re-compute an unbounded triangle's collapse time. Fortunately, this is not necessary: An unbounded triangle collapses if and only if its vertices v_1, v_2 and the north pole lie on a great circle. This is exactly when v_1, v_2 and the south pole lie on a great circle. Mapping things back to the plane, that means that an unbounded triangle collapses when the triangle $\Delta(v_1, v_2, o)$, where o is the origin, collapses. Computing the time when two kinetic vertices, v_1 and v_2, become linearly dependent is straightforward and integrates well with our handling of bounded triangles.



5 DEALING WITH REAL-WORLD INPUT

The algorithm as described by Aichholzer and Aurenhammer [AA98] has a non-obvious implicit assumption that the input is in general position. In the case of this algorithm general position means that no two events ever happen at the same time. When we tested our implementation during its development on real-world input, this resulted in problems.

In this chapter we will describe two issues that arise from this assumption as well as how we have extended the algorithm to work in these special cases. In particular, our solutions do not rely on exact arithmetic operations and instead work with finite-precision operations implemented in common computing hardware such as IEEE 754.

5.1 PARALLEL WAVEFRONTS

Consider the C-shaped planar straight-line graph depicted in black in Figure 19. Its emanated wavefront just prior to the first event is shown in red. The area not yet visited by the wavefront is triangulated in an arbitrary manner.

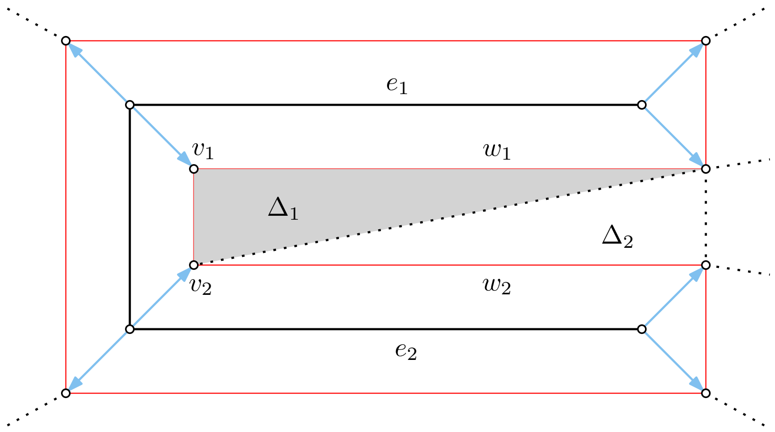
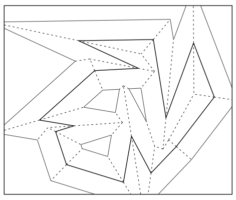


Figure 19: Vertices v_1 and v_2 collide in an edge event as Δ_1 collapses. At what speed and in which direction will the vertex that is created during event handling move?

As the wavefront propagates, the two wavefront edges w_1 and w_2 will crash into each other. Since they have been emanated from e_1



and e_2, the two legs of the letter C that are parallel, the wavefront edges themselves are parallel as well.

Colliding parallel wavefronts are always witnessed by more than one triangle collapse. This is easy to see: Before the collapse the two edges, w_1 and w_2, span a trapezoidal area. This area has not yet been visited by the wavefront and thus is covered by a set of triangles of the triangulation. The wavefront edges w_1 and w_2 have incident triangles Δ_1 and Δ_2. Since w_1 and w_2 share no common vertices, it follows that $\Delta_1 \neq \Delta_2$ must hold. Therefore, no less than two triangles cover, at least in part, the trapezoidal area spanned by w_1 and w_2. When this area collapses to zero as w_1 and w_2 collide, these triangles all collapse simultaneously.

In Figure 19 for instance, we observe triangles Δ_1 and Δ_2 collapsing when the two wavefronts collide. There also is a third, unbounded triangle to the right of Δ_2 but that is not relevant here so we shall ignore it.

Since there is more than one triangle collapsing at the same point in time, we also have more than one event happening simultaneously. It is unclear which of these events should be processed first; the original algorithm description gives no guidelines.

Assume we handle the collapse of the shaded triangle Δ_1 first. Then, as the triangle collapses, vertices v_1 and v_2 become incident and we have an edge event. As described in Section 4.5, edge events replace the two incident vertices with a new vertex v', which moves along the angular bisector of its two incident wavefront edges. The speed of v' is defined in such a way that it will keep up with the wavefront propagating at unit speed.

In the edge event that will happen in Figure 19 these two incident wavefront edges, however, are parallel and actually overlap at the time of the event. This implies that v' will have to move at infinite speed in order to "keep up" with the wavefront that is propagating perpendicular to it.

We proceed as follows: We start by handling the edge event similar to how we would handle an ordinary edge event, that is, we create v' and update incidence information as required. Then we set a special flag at the newly created kinetic vertex which indicates it is moving *infinitely fast*.

Like any other vertex, such an infinitely fast moving vertex v' is the central vertex of a triangle fan of one or more triangles $\Delta'_1, \Delta'_2, \ldots, \Delta'_n$. The fan at v' is enclosed by two overlapping wavefront edges — otherwise v' would not be infinitely fast. That implies that all triangles of the fan are collapsing at the time the infinitely fast moving vertex comes into existence.

We can, therefore, immediately handle the events witnessed by any of these triangle collapses. Among all these triangles we choose either Δ'_1 or Δ'_n, depending on which has the shorter wavefront edge to v'. Let Δ' be the triangle chosen and let v'' be the other vertex next to v' on the wavefront edge. If Δ' has two wavefront edges incident to v', then v'' shall be on the shorter wavefront edge.

In Δ' we process an edge event as if v' had become coincident with v'': We add the path from v' to v'' as a straight skeleton arc, and v' and v'' merge into a new kinetic vertex, leaving behind a straight skeleton node. If the newly created kinetic vertex is again infinitely fast, we repeat this process. Otherwise, the new kinetic vertex is an ordinary vertex and all incident triangles that were previously in the triangle fan will have positive area and collapse at a later time or never. Unless, of course, some other reason exists for them to collapse now in which case we process these events normally.

If in our example depicted in Figure 19 we would have chosen the other triangle, Δ_2, first, then this too would have resulted in an edge event creating an infinitely fast moving vertex and processing would have been identical.

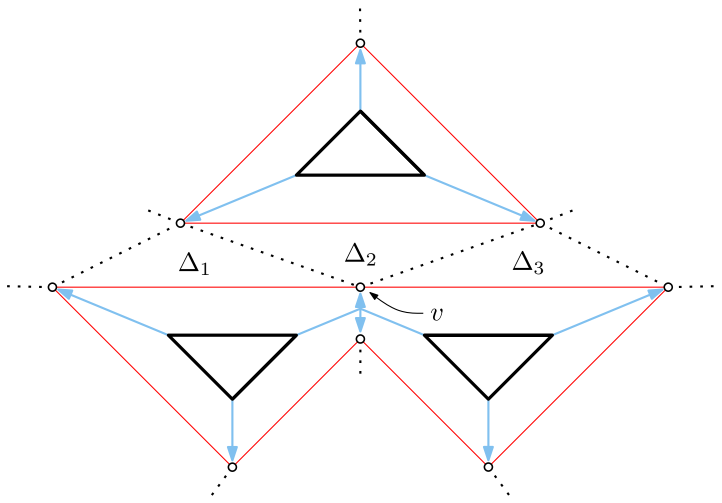
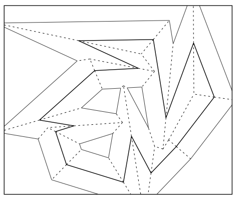


Figure 20: Infinitely fast moving vertices cannot only appear as the result of edge events.

Observe that infinitely fast moving vertices not only appear during edge events. The input graph, wavefront, and partial straight skeleton shown in Figure 20 provide such an example. When the two parallel wavefronts collide and triangles Δ_1, Δ_2, and Δ_3 collapse, we have a choice of split events. If we handle triangle Δ_2 first, where v crashes into its opposing wavefront, this will produce two infinitely fast vertices, one moving left, the other moving right. Processing happens similar to above, except that after the initial split event we will have



not one but two collapsing triangle fans and we handle both before processing any other event.

Note that this concept of infinitely fast moving vertices is not a result of any limited precision computation. The same issues will arise with exact geometric predicates and computation.

5.2 SIMULTANEOUS FLIP-EVENTS

For an algorithm to be useful some requirements have long been established. One of the more obvious ones is that upon termination the algorithm should return the correct result. This is called *partial correctness* in the literature. A slightly less obvious requirement is that of *total correctness*, meaning that the algorithm should terminate for any given input.

Unfortunately, the algorithm by Aichholzer and Aurenhammer does not ensure termination on arbitrary input. To see why that is, let us review the argument from Section 3.4.2 for why the triangulation-based algorithm terminates when the input is in general position. This argument is made by establishing bounds on the number of events the algorithm processes as follows:

- Every edge event and every split event reduces the number of triangles in the kinetic triangulation by exactly one. Edge and split events do not create new triangles and the flip event does not create more triangles than it removes. Therefore, the number of triangles is strictly decreasing with every edge or split event. This provides an upper bound on the number of edge and split events.

- We are not able to establish such a tight bound for flip events: Let n be the number of vertices in the input PSLG and let k be the number of kinetic vertices seen during the entire propagation process. We know that $k \in \mathcal{O}(n)$.

  There there can be at most $\binom{k}{3}$ different triangles since that is the number of possible combinations of three elements out of k.

  Kinetic vertices move at their constant speed along straight lines. Three points in the plane moving at constant velocity become collinear exactly once, twice or never. The area spanned by three such points is a quadratic polynomial in time — see Section 4.3. Therefore, a kinetic triangle of three such kinetic vertices will collapse at most twice.

  This implies the number of different collapse times leading to flip events is in $\mathcal{O}(n^3)$.

If an input is in general position, then no two events will ever happen at the same time. Consequently, the upper bound on different collapse times yields an upper bound on the number of flip events as well.

Therefore, the triangulation-based algorithm will terminate provided the input is in general position.

ARBITRARY INPUT. What problems do we incur if the input is not in general position? Obviously the bound on edge and split events still holds. Unfortunately however, the bound on different collapse times is no longer useful.

Consider the situation in Figure 21a. Vertices v_1 and v_2 move upwards and vertices v_3 and v_4 move downwards. When the wavefront propagation has processed a bit further in the near future, all four vertices will be collinear and triangles Δ_1 and Δ_2 will have collapsed. Let t be that time in the wavefront propagation process.

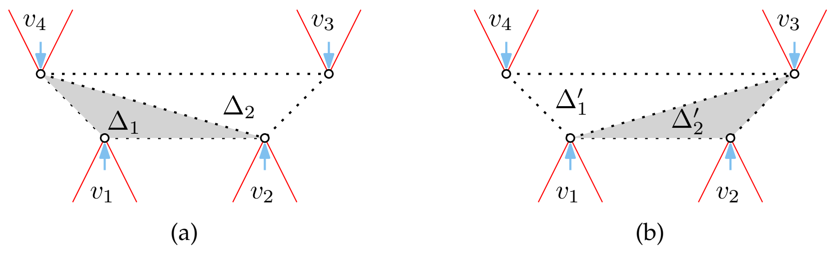
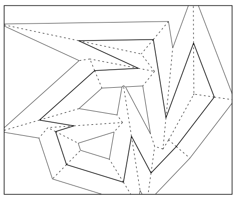


Figure 21: A potential flip-event loop: Processing the shaded triangle in Subfigure (a) first will result in the configuration from Subfigure (b). Again processing the event caused by the shaded triangle will bring us back to the initial triangulation.

At time t, when both Δ_1 and Δ_2 collapse, no two vertices become coincident. Furthermore, since neither triangle has a wavefront edge, we can conclude that each collapse indicates a flip event.

We have not established any guidelines on which event to process first when several events happen at the same time. Assume we choose to first process the collapse of the shaded triangle, Δ_1. This implies removing the spoke (v_2, v_4) and adding the spoke (v_1, v_3). Triangles Δ_1 and Δ_2 are replaced with new triangles Δ'_1 and Δ'_2. This situation is shown in Figure 21b. Since this procedure also destroys the triangle Δ_2, we don't have to process the event caused by the collapse of Δ_2.

However, we have just created two new triangles whose collapse times need to be computed in order to find out when they will trigger events and what kind of events those will be. Both Δ'_1 and Δ'_2 are constructed from the same four kinetic vertices that Δ_1 and Δ_2 had been



made of, namely v_1, v_2, v_3, v_4. These vertices are collinear at time t, the current time in the wavefront propagation process. Therefore, Δ'_1 and Δ'_2 will both collapse at time t also. Consequently, no progress is made in the wavefront propagation process with regard to time and we have to handle the events triggered by Δ'_1 and Δ'_2 now.

Again, we have two events that happen at the same time. Should we decide to process the flip event witnessed by the collapse of Δ'_2 now, we will end up in the very same triangulation as two events ago, in Figure 21a.

This example shows that it is possible to end up in an infinite loop of flip events, proving that the algorithm cannot satisfy the requirements for total correctness.

A DIFFERENT PROCESSING ORDER RESULTS IN PROGRESS. Note that if in either situation we would have picked the other event first, we would have broken the loop:

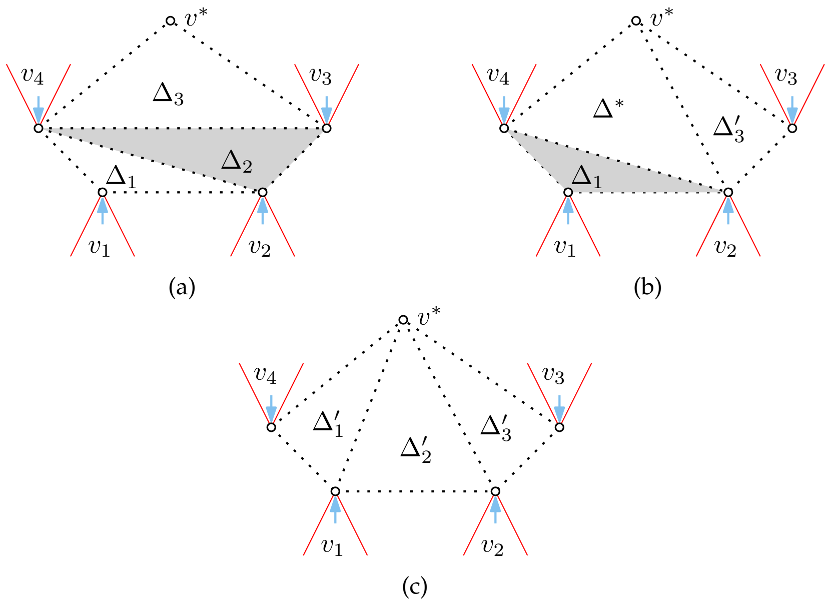


Figure 22: A potential flip-event loop averted. By flipping the correct triangle first, we ensure that we make progress.

Let us consider the same scenario again. In Figure 22a we see the same initial triangulation, this time including Δ_2's neighbor along (v_3, v_4). In order to better see the triangulation, we have once more drawn the vertices in their positions just prior to becoming collinear. This is only for demonstrative purposes however; the events we are considering happen when v_1, v_2, v_3, v_4 are all on a common supporting line.

Contrary to the previous processing order, we now start with the flip event witnessed by Δ_2. The longest edge in the triangle is (v_3, v_4) and the vertex v_2 moves over that spoke. So we replace (v_3, v_4) with the spoke (v_2, v^*), remove Δ_2 and Δ_3 from the triangulation and add Δ^* and Δ_3' instead (see Figure 22b). Since these triangles have a non-zero area at time t, they are not collapsing right now.

We still have to deal with Δ_1. Again, we remove the triangle's longest edge, (v_2, v_4) this time, and replace it with the other diagonal in the remaining quadrilateral, that is, (v_1, v^*). The new triangles thus created are labeled in Figure 22c as Δ_1' and Δ_2'. These two again have positive area. Therefore, they will collapse at a point in time different from t.

Vertices v_1 through v_4 can continue on their original heading and we can continue with the wavefront propagation process.

In this different order of event processing the fact that two triangles collapsed simultaneously did not result in any complications.

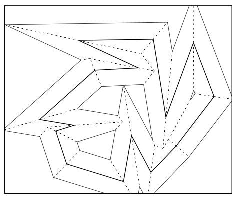
The obvious questions thus are how to detect potential flip-event loops and how to avoid or break out of such loops.

DETECTION. One of the first intuitions might be that as soon as one re-introduces a triangle previously deleted or sees the same event a second time, one must be caught in a flip-event loop. Unfortunately this is not the case.

Consider Figure 23: Let v_1 and v_6 move downwards and $v_2, \ldots, v_5$ upwards such that all six vertices lie on the same supporting line at some point in time, t, without any two vertices becoming incident. At time t all triangles shown in the triangulation will have degenerated to line segments on the same supporting line and as such all these triangles trigger events. All these events will be flip events and handling one such event means we have to replace the longest edge s of the witnessing triangle Δ with the other diagonal of the quadrilateral spanned by Δ and its neighbor along s.

In Figure 23(i) we show the initial triangulation. The triangle whose event we process first is indicated by shading and its flipping spoke is shown dotted. Executing this flip event will lead to the situation in (ii). We then repeatedly pick one of the collapsed triangles and process their event.

Particularly note the step from (iv) to (v) that re-introduces the triangle $\Delta(v_1, v_2, v_3)$, which was previously present in (i). We indicate this triangle as hatched. Also observe that at step (viii) we process the same event as in (i), namely triangle $\Delta(v_1, v_2, v_3)$ flipping along (v_1, v_3) into triangle $\Delta(v_1, v_3, v_4)$.



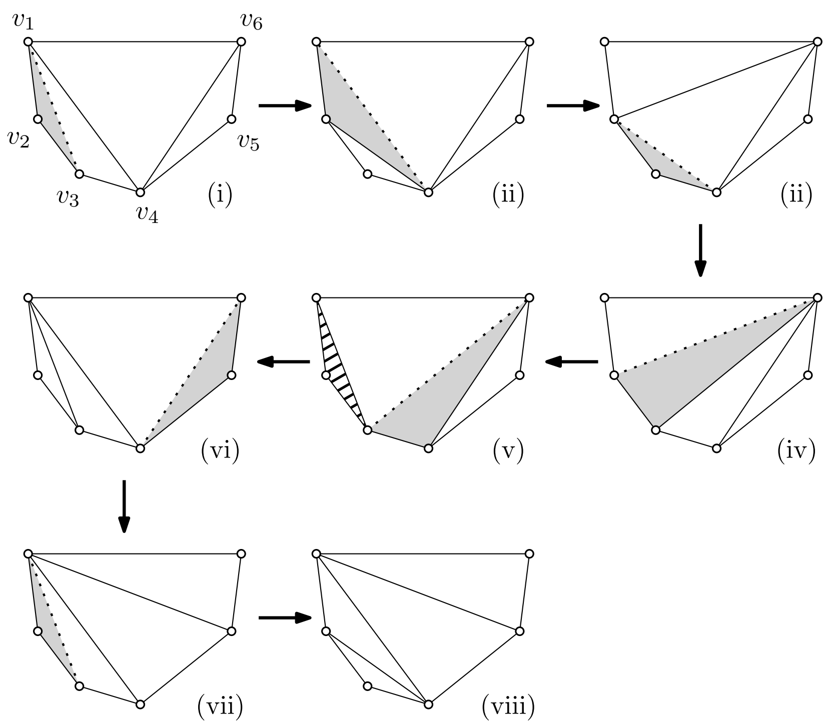


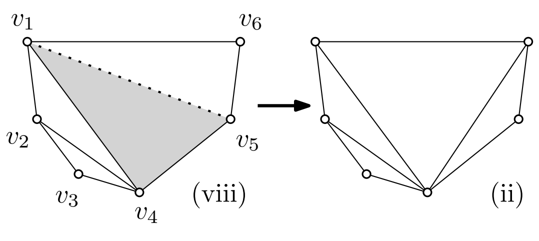
Figure 23: Processing the same flip event twice does not necessarily imply we are in a flip-event loop.

Note that no two triangulations shown in Figure 23 are identical, meaning that despite re-introducing triangles or repeatedly handling the same event we are not stuck in an event processing loop.

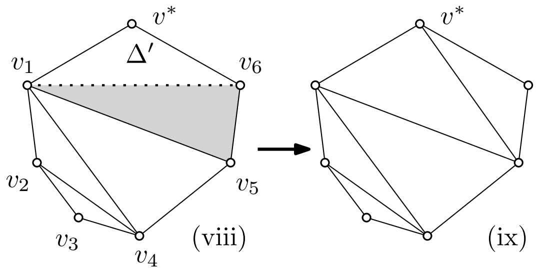
However, if we continue the processing as shown in Figure 24a we end up in a situation we have seen before, namely (ii). Assuming we have a deterministic algorithm which, given a triangulation with more than one collapsed triangle, always picks the same event to process first, we are now truly stuck in an eternal flip-event loop.

Conversely, if we select the top triangle, $\Delta = \Delta(v_1, v_5, v_6)$, as the one we want to process next, then we flip towards the outside of the collapsed region — see Figure 24b. Let Δ' be the neighbor of Δ along (v_1, v_6) and let v^* be the vertex of Δ' that is not v_1 or v_6. This vertex is not collinear with $v_1, \ldots, v_6$. Since Δ' has positive area, the flip event will make real progress and we have averted this particular flip-event loop.

We have shown that the detection of flip-event loops is not as simple as looking for whether we process the same event a second time, or re-introduce a triangle. A procedure that would obviously work is to compare a flip event's resulting triangulation with all previous triangulations. If we have arrived at an identical triangulation and our selection process is deterministic, we are stuck in a loop. However, comparing complete triangulations is an expensive process.



(a)



(b)

Figure 24: Depending on which triangle we process next, we (a) either end up in a loop or (b) we avoid being stuck in a flip-event loop.

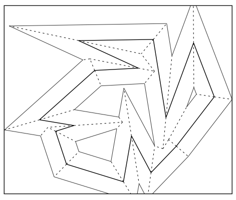
Before we return to the topic of detecting loops in Section 5.2.2, let us first discuss a method to avoid them entirely if exact arithmetic operations are available.

5.2.1 *Avoiding Flip-Event Loops with Exact Arithmetic*

Implementing the triangulation-based algorithm on a system that affords us exact arithmetic operations provides us with the following capabilities:

- We know the exact collapse time t for any triangle.
- In particular, if more than one triangle collapses at a specific time t, we can produce an exhaustive list of all triangles that collapse at t.
- We can always correctly determine the type of an event since we can tell exactly whether two vertices have become incident, indicating an edge event, or not, indicating a split or flip event.
- We can compare lengths of spokes of triangles at a time t and correctly identify the longer one.

With these capabilities we can devise an algorithm that will tell us which event to process first if more than one happens at the same



time. This algorithm will have the property that it avoids flip-event loops inherently, without having to spend any effort on detecting and then escaping them.

Algorithm 3 Decide which event to handle next

```
1:  procedure CHOOSE_NEXT_EVENT($E$)
                              ▷ $E$ ... non-empty set of events happening now.
2:      if $E$ contains non-flip events then
3:          Choose an arbitrary non-flip event $c \in E$.
4:          return $c$.
5:      else if $E$ contains only flip events then
6:          Let $s(e)$ be the flipping spoke of an event $e$.
7:          Let $|s(e)|$ be its length.
8:          Choose $c \in E$ such that $|s(c)| \geq |s(e)|$ for all $e \in E$.
                              ▷ $e$ is the flip event with the longest flipping edge.
9:          return $c$.
10:     end if
11: end procedure
```

Why does Algorithm 3 provide this property? Let E be the set of events that are happening right now because their corresponding triangle collapsed. We already have classified them and we can now distinguish between two cases: (i) there are edge or split events in E and (ii) E consists entirely of flip events.

If E contains edge or split events, then picking any such event guarantees progress. Recall that processing a non-flip event will reduce the number of triangles in our kinetic triangulation by one. Since the number of triangles never grows, a monotonicity argument shows that this moves us forward — we can never again end up in the state prior to having processed this event.

In the other case, E consists only of flip events. We will again construct a monotonicity argument.

Let $\Delta(e)$ be the collapsed triangle corresponding to an event $e \in E$, and let $s(e)$ be the flipping spoke of e. According to the rules of flip events, $s(e)$ is the longest edge of $\Delta(e)$. Since e is a flip event and not an edge event, we know that $s(e)$ is unique in $\Delta(e)$. Furthermore, let $\Delta(E) = \{\Delta(e) | e \in E\}$ be the set of all triangles collapsing right now.

Chose event $c \in E$ such that $|s(c)|$, the length of $s(c)$, is maximal for all events. Let Δ^* be the neighbor of $\Delta(c)$ along $s(c)$.

- If $\Delta^* \notin \Delta(E)$, then this triangle must have positive area. Therefore, none of its vertices are coincident. We also know that none of the vertices of $\Delta(c)$ vertices are coincident since this collapse triggered a flip event and not an edge event. Thus, flipping $s(c)$ to the other diagonal of the quadrilateral spanned by $\Delta(c)$

and Δ^* will remove these two triangles from the kinetic triangulation and add two new triangles with positive area. This means that they will collapse not now but at a later time, if at all. We have reduced the number of triangles collapsing at this particular time and have, therefore, made progress.

- In the case of $\Delta^* \in \Delta(E)$ the argument is slightly different: The triangle whose event we are processing right now is $\Delta(c)$. Let its vertices be v_1, v_2, v_3 and let the vertices of Δ^* be v_1, v_3, v^*. Note that Δ^* is collapsing as well. We know that all four vertices, v_1, v_2, v_3, and v^*, are collinear and we know that v_1, v_2, v_3 are pairwise distinct and v_1, v_3, v^* are as well.

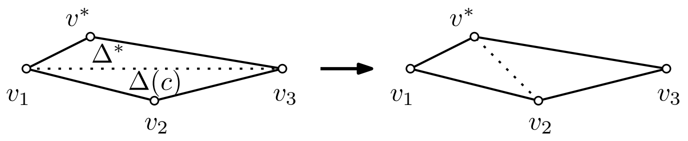


Figure 25: Flipping the longest edge between two collapsed triangles will result in a shorter edge separating the new triangles.

The longest edge of any collapsing triangle is $s(c) = (v_1, v_3)$. Therefore, v_2 is on the line segment between v_1 and v_3 and, likewise, v^* is in between v_1 and v_3. (The vertices v_2 and v^* may or may not be coincident — it does not affect our argument.) This implies that the length of (v_2, v^*) is less than the length of $s(c) = (v_1, v_3)$, that is, $|(v_2, v^*)| < |s(c)|$. (See Figure 25.)

Therefore, replacing $s(c)$ with (v_2, v^*) will reduce the length of the longest edge of the set of collapsing triangles or, if there were more than one edge of maximal length, it will reduce the number of edges that have maximal length.

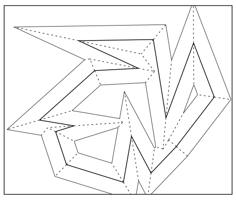
Note that since this processing happens at a fixed time t, all kinetic vertices have a constant location on the plane. This implies that there are only a fixed, finite number of possible edges and discrete edge lengths. Thus, the property just presented can be used to support a monotonicity argument.

Summarizing, the procedure outlined in Algorithm 3, ensures that we either:

(i) reduce the number of triangles in the kinetic triangulation overall if there are non-flip events,

(ii) reduce the number of triangles currently collapsing if we flip towards a non-collapsing triangle, or

(iii) reduce the number of edges of maximal length or reduce the maximal length over all edges in case we flip within the col-



lapsed triangles, by replacing an edge of maximal length with a shorter one.

Combined, these ensure that after a finite number of steps we have processed all events at a particular collapse time t and can proceed in the wavefront propagation process.

Observe that this approach does not add any additional run-time complexity. The sort key for our priority queue, which so far has only consisted of the collapse times of triangles, can easily be extended to be a multi-value key which includes factors that determine ordering of events that happen at the same time, namely event-type and length of flip edge, if applicable. This will ensure events are processed in an order which guarantees the algorithm will terminate.

5.2.2 *Flip-Event Loops with Finite Precision*

Unfortunately for our implementation, we cannot make use of the method described in the previous section. Our implementation can use either standard IEEE 754 double precision floating-point operations or, if requested by the user, MPFR's arbitrary precision arithmetic [FHL+07].

Due to rounding errors introduced by finite-precision computation, correct ordering of events—or even establishing that several events happen at the same time—is very error prone. We therefore have developed the following procedure, first described in [PHH12], to detect and resolve potential flip-event loops.

DETECTION. We keep a journal J of tuples (t, Δ) where t is a point in time in the wavefront propagation process and Δ is a triangle as specified by its set of three kinetic vertices.

For every flip event e that we process, we append to J the tuple $(t(e), \Delta(e))$. As before, $\Delta(e)$ is the collapsed triangle corresponding to e, and $t(e)$ is its collapse time. When we process an edge or a split event, we clear the journal. Therefore, J is a list of of flip events since the last non-flip event, sorted sequentially by processing order.

Additionally, we maintain a search structure S that allows us to find a given vertex triple Δ in J.

Whenever we process a flip event e, we consult S to check whether $(t(e), \Delta(e))$ is in our journal already. If we are in a flip-event loop, then this must be the case. As discussed previously (see Figure 23), the converse is not necessarily true. We are unable to tell if we are truly in a flip-event loop but regardless, if we find the event has been processed once before we conduct the following resolution procedure. No damage is caused in case of a false alarm.

RESOLUTION. Let T_1 be the first occurrence of $(t(e), \Delta(e))$ in J and let T_2 be the occurrence we just inserted. If we had exact arithmetic operations, then the time components of T_1 and T_2 would be the same and we could infer that the clock had not progressed between these two events. Therefore, the time components of all flip events between T_1 and T_2 would have to be identical. On a finite-precision system we have no such guarantee as rounding errors might have introduced slight variations in the time component. Nevertheless, we declare that all these events happened at the same time.

Next, we revert all the events logged in J starting at T_2 all the way back to T_1. Note that since only flip events are recorded in J, this will not bring back any triangles already removed, annihilate any kinetic vertex just created or resurrect any old kinetic vertices. During the revert steps we mark all triangles that caused a flip event as having collapsed. We also mark all other triangles for which we can infer they have collapsed from how they were involved in flip events. We know that the set of collapsed triangles so marked will form one or more polygons which each have degenerated to straight-line segments.

Let P be the polygon that contains $\Delta(e)$. First, we replay all the flip events of triangles not in P — they are not part of the potential loop we try to resolve here. Care must be taken to correctly update J and S during the revert and replay operations.

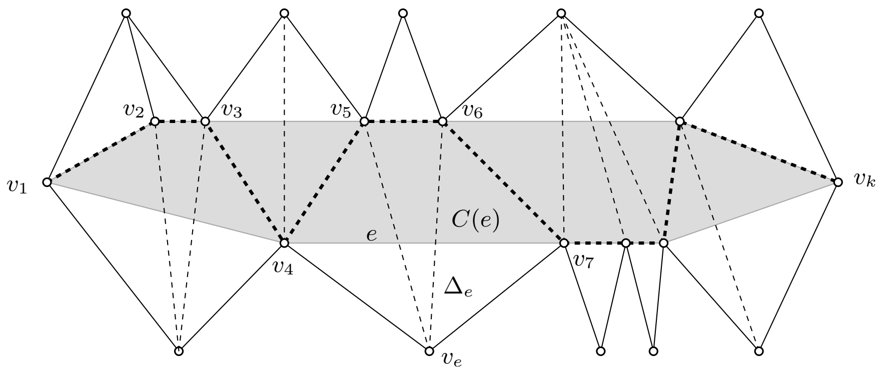
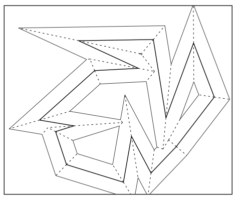


Figure 26: Re-triangulating a collapsed polygon as shown is one step of our flip-event-loop resolution procedure.

The polygon P has collapsed to a straight-line segment. Let l be the supporting line of that segment and let $v_1, v_2, \ldots, v_k$ be the vertices of P, sorted along l. We construct a path $p = (v_1, v_2, \ldots, v_k)$. Since the vertices v_i are sorted along l, this path will be monotone with respect to l. We update the triangulation of P such that all edges (v_i, v_{i+1}) of p are triangulation spokes within P.



This will partition P into a set of faces that are each bounded by exactly one external edge e on the outside and edges of p on their remaining sides.

Next, we also consider all triangles Δ_e which are neighboring P along edges e which are not part of p. Consider a particular edge e and its incident outside triangle Δ_e as shown in Figure 26. Let v_e be the vertex opposite of e in Δ_e. We re-triangulate in such a way that the area consisting of Δ_e and the polygon face bounded by e and p becomes a triangle fan about v_e. In our example this polygon is made up of the vertices v_4, v_5, v_6, v_7, and v_e.

We require that by following the events logged in our journal J one arrives at the target triangulation which we just described. One option is to re-triangulate P and its neighbors without taking the existing triangulation into consideration and then apply an algorithm described by Hanke et al. [HOS96]. This algorithm produces, given two different triangulations of the same point set, an order of flips to transform one triangulation to the other. We could run this algorithm and append the flip sequence to J.

In practice we observed that a naive incremental algorithm works satisfactory: Using again the example polygon in Figure 26 we start by building a triangle fan of the first face centered at v_1. We do this by repeatedly flipping the edge (v_2, v_i) of the triangle incident at (v_1, v_2) until (v_1, v_3) is a triangulation spoke. We then continue to repeatedly flip (v_3, v_i) of the next triangle incident at (v_1, v_3) and so on. Whenever we get to a vertex that "changes sides" (such as v_4 in our figure), we have to flip away any extra spokes incident at the triangle fan's center. When no extra spokes are left, we will have completed this face. We then start a new triangle fan for the next face. Once we have adapted the triangulation of P to our requirements using just flip operations, we can likewise move the center of each triangle fan to the v_e vertices outside of P using straight forward flips.

The flips we are executing in order to arrive at our desired triangulation are recorded in J in the same manner as any flip due to a flip event would be, with one exception: Let T^* be the first entry we add in the reconfiguration procedure. All records we subsequently append will have a pointer to T^*. This allows us to infer just by looking at the journal that all these events happened at what we consider to be exactly the same moment. Its purpose will become apparent shortly.

We resume with normal event processing after we re-triangulated P and its neighbors and J and S were updated accordingly.

Assume now that one of the triangles Δ_e, that were adjacent to P and that we extended our triangulation into, also had collapsed at the same time and v_e was collinear with P. This fact was not known when we previously applied our resolution procedure because that triangle had not shown up in the loop. If our loop detection subsequently triggers because we are processing an event whose tuple T is already in the journal, we proceed as follows: If T has no back-pointer, we process the loop normally as just described. If, however, T has a back-pointer to a T^*, indicating it was part of a flip-event resolution attempt once before, we change the procedure slightly. Instead of rolling back to just T, we roll back all the way to T^* and run the resolution algorithm from there. This ensures that the new flip polygon P' includes the old polygon P and we have even more vertices that we know are collinear. Since there are only finitely many triangles, enlarging the flip polygon each time we run into the same loop again ensures we will eventually find the maximal polygon. This polygon will resolve properly with the procedure described.

FINDING THE COLLAPSED POLYGON P. We mentioned that while reverting the journal we mark triangles that are involved in flip events to find find a set of triangles which have collapsed to a straight-line segment. We describe this process in more detail now.

We start the revert process with having none of the triangles in the kinetic triangulation $\mathcal{K}$ being tagged.

Consider a flip event e where a triangle $\Delta_1 = (v_1, v_2, v_3)$ collapses (Figure 27). Let $\Delta_2 = (v_1, v_3, v_4)$ be its neighbor along its longest edge, that is, Δ_1 will flip into Δ_2. Both triangles get removed by that flip event and replaced with triangles $\Delta_1' = (v_1, v_2, v_4)$ and $\Delta_2' = (v_2, v_3, v_4)$.

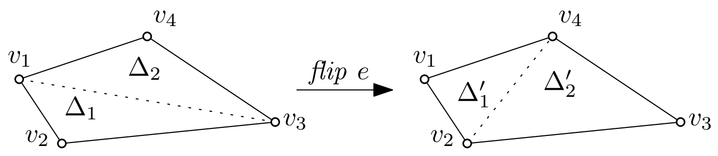
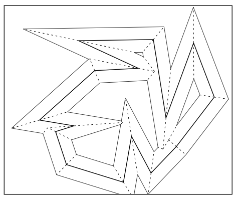


Figure 27: Flip event e replaces triangles Δ_1 and Δ_2 with Δ_1' and Δ_2'.

Now consider how to roll back e: Δ_1' and Δ_2' are both in $\mathcal{K}$ prior to the revert. If neither is tagged, we undo the flip, and tag Δ_1 which we know has collapsed—it caused the flip event after all (Figure 28a).

If, however, Δ_1' or Δ_2' (or both) are already tagged, then after the revert we tag both Δ_1 and Δ_2 (Figure 28b). Why is this correct? Without loss of generality let us assume that Δ_1' was tagged before the revert, and thus vertices v_1, v_2, v_4 are known to be collinear. Since Δ_1 has collapsed, vertices v_1, v_2, v_3 are also collinear. This implies that all four vertices, v_1, v_2, v_3, v_4 are collinear because $v_1 \neq v_2$. Therefore, we can tag both triangles, Δ_1 and Δ_2.



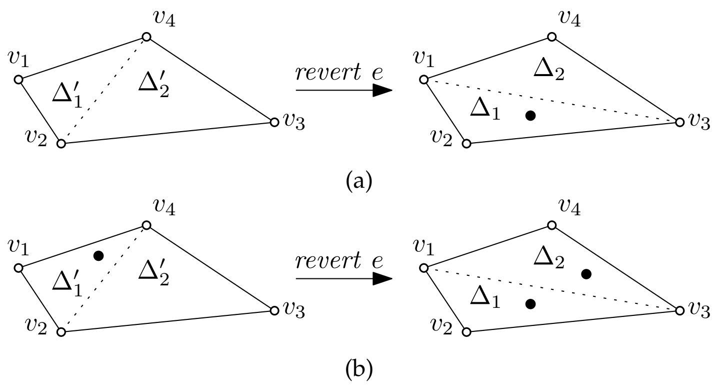


Figure 28: Reverting e: The • symbol indicates whether a triangle is tagged, that is, we know it has collapsed. (a) If prior to the revert neither triangle is tagged, then only the one having caused the event gets tagged after the revert. (b) If previously either triangle is tagged, then both triangles are subsequently marked.

We construct P incrementally by adding tagged neighbors to a growing set of triangles, in reminiscence of how flood-fill works: Let Δ be the triangle that we saw twice in J and that caused us to do this resolution procedure. Then we start with $P_0 = \{\Delta\}$. In the incremental step, we set $P_{i+1} = P_i \cup \mathcal{N}(P_i)$ where $\mathcal{N}(P_i)$ is the set of triangles in $\mathcal{K}$ that are both tagged and that share an edge with a triangle already in P_i. This process ends once P_i no longer gets larger, i.e. when $P_{i+1} = P_i$ for a particular i. Then $P = P_i$ and all vertices of P are collinear.

WAVEFRONT EDGES ON P. Recall how we extended the triangulation of P to incorporate the areas spanned by triangles adjacent to P along edges e. These edges were the the border edges of the polygon minus those edges covered by p, the monotone path we constructed from v_1 to v_k.

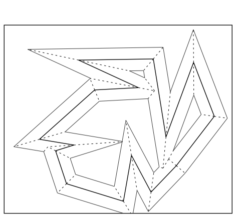
During the reconfiguration we may discover that one such edge e may not be a triangulation spoke after all but in fact a wavefront edge. As such, there would be no adjacent triangle Δ_e to extend the triangulation into.

Let $e = (v_i, v_j)$ with $i < j$. As e is not on p, there must be a v_m with $i < m < j$. That is, v_m is in between v_i and v_j on their supporting line, on the other side of the polygon.

We can, therefore, schedule an immediate split event where v_m splits the wavefront edge (v_i, v_j). Since split events reduce the number of triangles in the overall triangulation, we again have a guaranteed progress of our algorithm.

SORTING VERTICES ALONG THE COLLAPSE LINE. In order to build p, we need to sort the vertices along P's supporting line. If we had exact arithmetic operations, then all the vertices would indeed be on the exact same line and all collapse times would have been identical. Due to rounding errors when using finite-precision operations however, we may have slight variations in collapse times and not all vertices will line up exactly.

To arrive at a sort order, we proceed as follows: Let t_{min} and t_{max} be, respectively, the minimum and maximum collapse time of the triangles in P. We then determine a fitting straight-line L_{min} of the vertices $v_1, \ldots, v_k$ at time t_{min} using a least-square fitting. We construct L_{max} accordingly. Next, we sort the vertex set twice. Once with respect to L_{min} and once with respect to L_{max}. If the two orders agree no special handling is necessary and we proceed as discussed previously. If the two sort orders do not agree, then there will be at least two vertices, v_i and v_j, whose relative order has changed. Using exact operations these vertices would in fact be coincident. We update the triangulation to enforce a spoke s between v_i and v_j and then force an edge event at both triangles incident to s. Consequently, the number of triangles in the triangulation is decremented and any ongoing loop is broken.



6 EXPERIMENTAL RESULTS

The upper bound on run-time complexity for the triangulation-based algorithm has been discussed in Section 3.4.2. Of course, these are theoretical, worst case bounds. We studied the actual count of flip events in real applications of this algorithm. Furthermore, we investigated the number of affected triangles in edge and split events.

We extended Surfer with extensive tracing code which can provide us with various data points about our implementation's behavior for a given input. In particular, we count the number of events of each type: *edge-*, *split-*, and *flip*-event. We further differentiate between kinds of edge events: (i) cases where all three edges of a triangle collapse at the same time, (ii) collapsing triangles with two wavefront-edges, and (iii) other edge events.

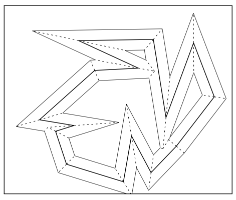
We tested Surfer on about twenty thousand polygons and PSLGs, with up to 2.5 million vertices per input. Both real-world and contrived data of different characteristics was tested, including CAD/CAM designs, printed-circuit board layouts, geographic maps, space filling curves, star-shaped polygons, and random polygons generated by RPG [AH96], as well as sampled spline curves, families of offset curves, font outlines, and fractal curves. Some datasets contain also circular arcs, which we approximated by polygonal chains in a preprocessing step.

6.1 EDGE AND SPLIT EVENTS

In theory, each edge or split event can affect as many as $\mathcal{O}(n)$ triangles. As mentioned in Section 3.4.2, Huber presented a convex polygon that, with a given triangulation, forces $\Omega(n)$ many triangles to be updated $\Omega(n)$ many times. See [Hub12, page 48] and Figure 10.

In practice, we have observed few, but still some, cases where a single edge event affects thousands of other triangles. The percentage of our test inputs which cause such extreme edge events is relatively small however. See Figure 29 and Figure 30.

Even in inputs that have such expensive edge or split events the total number of expensive events appears to be small. Indeed, we have found that, on average, the count of triangles we have to update is around one or two and well below ten for almost all inputs tested. We



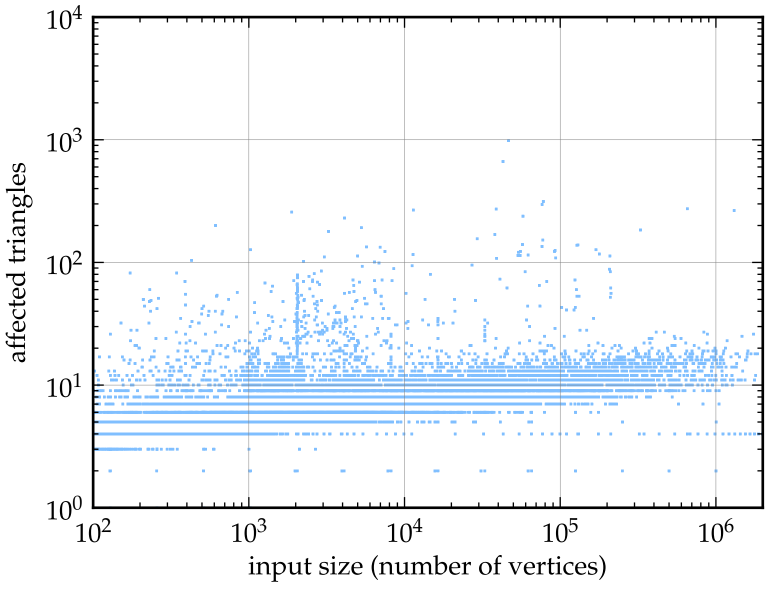


Figure 29: Maximum number of triangles affected by a single edge event.

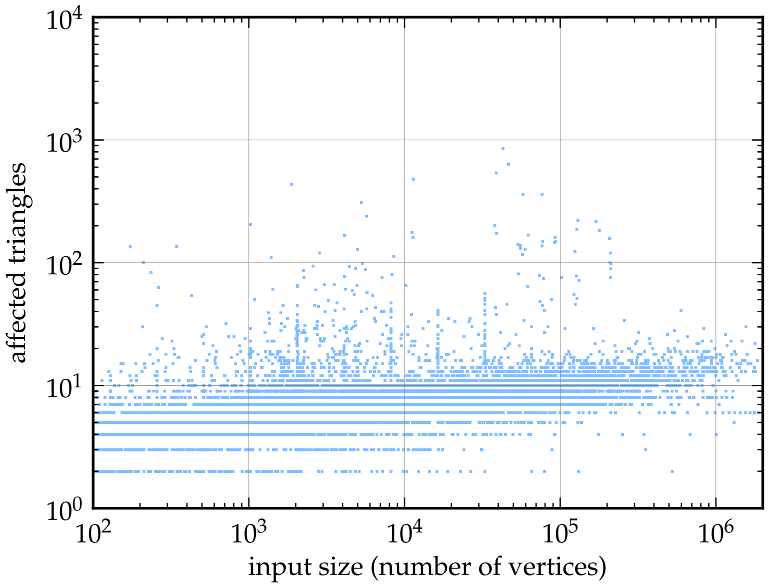


Figure 30: Maximum number of triangles affected by a single split event.

show this in Figure 31, where we plot number of affected triangles per non-flip event on the y-axis against different input sizes on the x-axis.

There still are, however, occasional outliers where we have to update more than just a single digit number of triangles per event. A cursory investigation suggests that densely sampled arcs or line segments are the prime cause for both, high maximum affected triangles per single event and high average affected triangles. In contrast, random polygons seem to be generally well behaved.

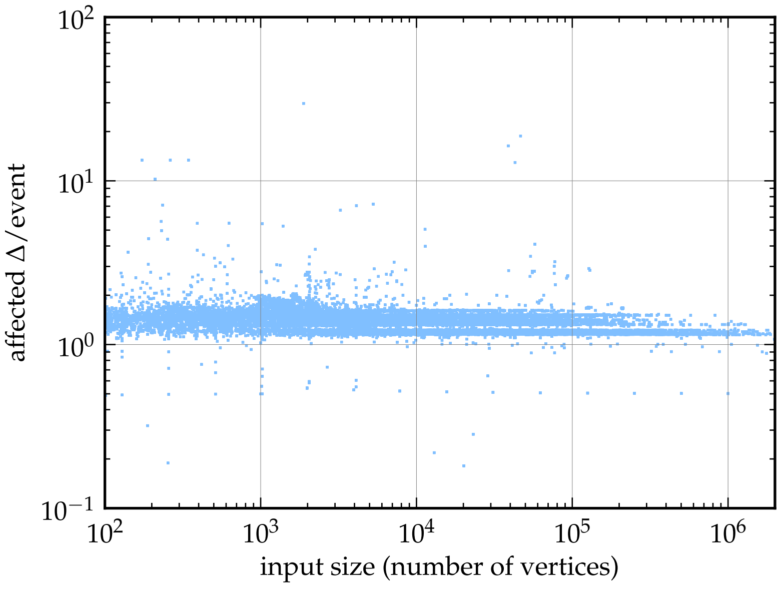


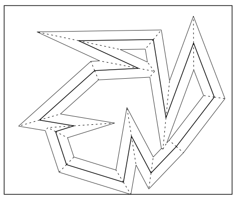
Figure 31: Average number of affected triangles per edge or split event

6.2 FLIP EVENTS

The upper bound on flip events is $\mathcal{O}(n^3)$ when input is in general position. There also is a known lower bound of $\Omega(n^2)$ forced by a particularly contrived input. See Section 3.4.2 and Section 5.2. The theoretical cost of handling flip events dominates all other processing in the algorithm. Therefore it is of particular interest how this number behaves in practice.

Our experiments provide strong experimental evidence that we can indeed expect a linear number of flip events for all practical data. In Figure 32 we show the number of flip events per input vertex for different input sizes n arranged on the x-axis.

On average, Surfer had to deal with a total of $n/4$ flip events. Over the entire set of twenty-thousand inputs only a dozen cases, mostly



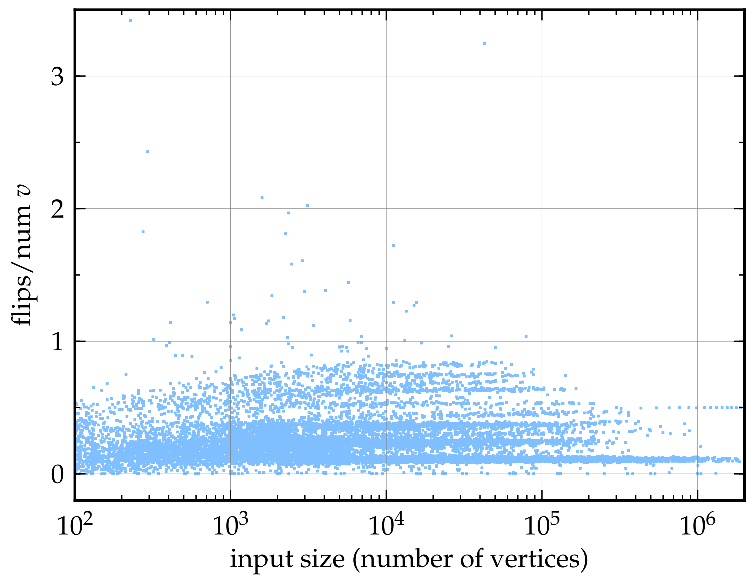


Figure 32: Number of flip events, normalized to input size.

sampled arcs, required more than $2n$ flip events. This clearly demonstrates the linear nature of this number in practice.

It is interesting to note the clusters in this plot. Some clusters, but not all, correspond to different types of input. For instance, a closer inspection of the test results revealed that synthetic "random" polygons generated by RPG require significantly more flips than random axis-aligned polygons.

6.3 RUNTIME BEHAVIOR

In the previous sections we provided experimental evidence that, on average, the number of flip events is linear in the input size and that the number of triangles affected by edge and split events is, again on average, constant. This suggests that in practice the overall running time of the algorithm should approximately follow an $n \cdot \log n$ law.

We conducted timing tests of Surfer using a 64-Bit Linux system running on an Intel Core i7-980X CPU clocked at 3.33 GHz. Surfer was compiled by GCC 4.4.3.

By default, Surfer uses standard IEEE 754 double-precision floating-point arithmetic, but it can be built to use the MPFR library [FHL+07, MPF], enabling extended-precision floating-point operations. When using floating-point arithmetic, it computes the straight skeleton of inputs with a million vertices in about ten seconds. In particular, our tests confirm an $\mathcal{O}(n \log n)$ runtime for practical data, including any time spent on handling degenerate cases.

COMPARISON TO OTHER SKELETONIZERS. We compared the runtime of Surfer against both Bone, the fastest other known implementation of a straight skeleton algorithm by Huber and Held [HH11], and against Cacciola's implementation [Cac12] that is shipped with the CGAL library, version 4.0 [CGA]. Input to the latter was confined to polygonal data as the implementation cannot handle generalized PSLGs. All three codes are single-threaded.

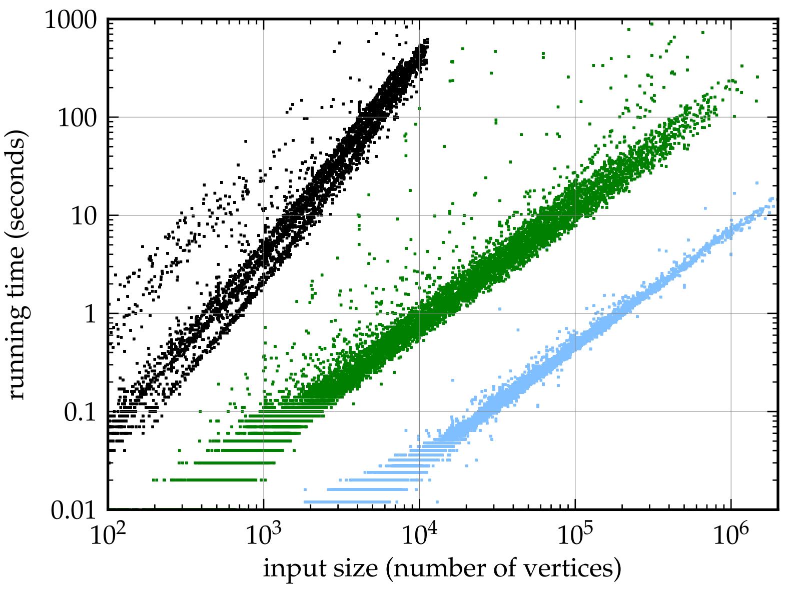
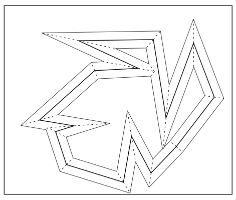


Figure 33: Runtime comparison between CGAL (black), Bone (green), and Surfer (blue).

As can be seen in Figure 33, Surfer consistently outperforms Bone by a factor of about ten. Furthermore, it is by a linear factor faster than the CGAL code. In particular, for inputs with 10^4 vertices CGAL already takes well over one hundred seconds whereas Surfer runs in a fraction of one second. Note, though, that the CGAL code uses its recommended exact-predicates-inexact-constructors kernel and, thus, could be expected to be somewhat slower. However, its timings do not improve substantially when run with an inexact kernel.

Further analysis revealed an average runtime (in seconds) of $5.8 \cdot 10^{-7} n \log n$ for Surfer, $1.5 \cdot 10^{-5} n \log n$ for Bone, and $4.5 \cdot 10^{-7} n^2 \log n$ for the CGAL code.



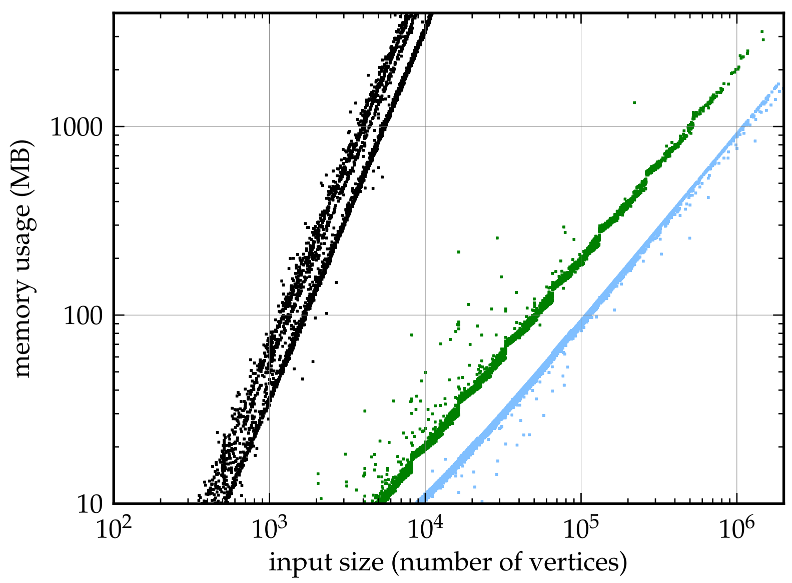


Figure 34: Memory usage comparison between CGAL (black), Bone (green), and Surfer (blue).

Figure 34 shows our measurements of memory consumption. These confirm the expected linear memory footprint of Surfer. Its memory usage is similar to that of Huber and Held's Bone, while the CGAL code, due to its algorithmic design, exhibits a worst-case quadratic behavior.

6.4 PHASES OF THE ALGORITHM

We analyzed Surfer's run-time behavior to learn where it spends most of its time. For this purpose we have divided a run of the algorithm into the following phases:

(1) pre-processing: In this step Surfer scales and translates the input graph such that all vertices lie in a square of edge length two centered at the origin.

(2) triangulation: The normalized input graph is handed over to the triangulation subroutine. In Surfer's current version that is Shewchuk's `triangle` [She96].

(3) kinetic triangulation: Once we have a constrained triangulation of the input, we set up the initial wavefront and the kinetic triangulation $\mathcal{K}$.

(4) initial schedule: We compute the collapse times of all triangles in $\mathcal{K}$ and do a preliminary classification of the event type each collapse will cause. We then set up the initial priority queue.

(5) propagation process: In this phase we simulate the wavefront propagation process. We handle events as they occur and update the wavefront's topology accordingly.

(6) post-processing: When the wavefront propagation has finished, we extract the straight skeleton from the set of wavefront vertices.

From our test runs on the set of twenty thousand inputs we selected runs with running time in excess of one second. For those we considered time spent in each of the six phases in relation to total run-time.

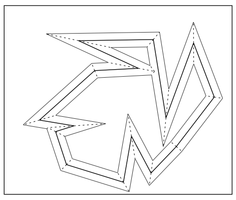
Our analysis shows that, on average, approximately a quarter of total run-time is spent in the triangulation code and a bit over half is needed for the actual simulation of the wavefront propagation process. Furthermore, we observe that there are a few inputs where most of the computation time is spent creating the initial triangulation. We have shown two samples in Appendix A: Figure 46 and Figure 47. The four other phases, pre-processing, setting up the kinetic triangulation and initial schedule, and post-processing vary only lightly. The box plot in Figure 35 summarizes the results.

6.5 RUNNING WITH EXTENDED PRECISION

Surfer can be built to use the MPFR library [FHL+07, MPF]. This library enables our code to use extended-precision floating-point operations, with a number of significant digits far in excess of the 52 bit generally available when using standard IEEE 754 double-precision arithmetic.

Obviously, higher precision arithmetic incurs a certain penalty both in runtime and space requirements. We benchmarked the performance and memory footprint of Surfer when using the MPFR backend at different precisions.

In Figure 36 we plotted the runtime of Surfer at different precisions, normalized to the runtime when using standard IEEE 754 floating-point arithmetic. Our data shows that for instance running at an MPFR precision of 100 takes about ten times as long as running in IEEE 754 mode; at a precision of 1000 the slow-down is already 25 and it is 110 at 4000 bits. The slow-down seems to follow an $n\sqrt{n}$ law due to the increased complexity of doing multiplications with a larger number of digits.



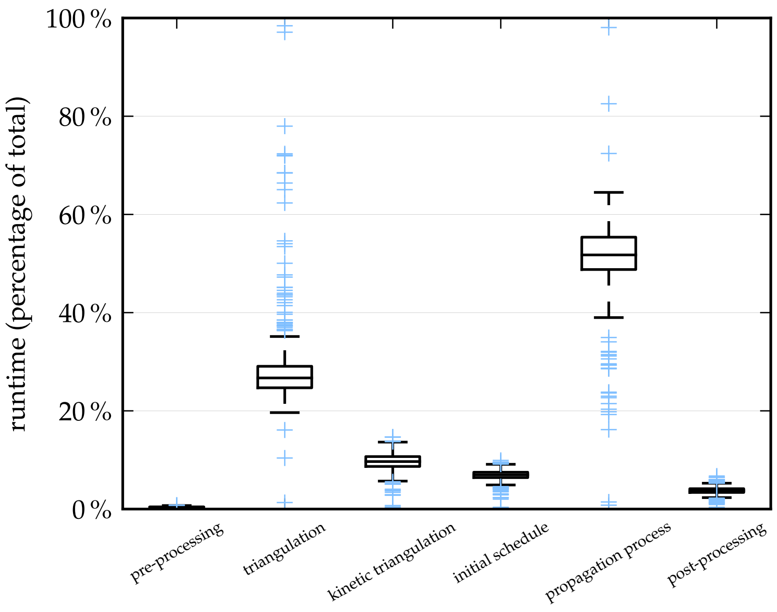


Figure 35: Fraction of time spent in each phase of Surfer. In this plot, a box is drawn between each set's first and third quartile. A horizontal bar is shown at the median. Whiskers are extended up and down from the third and first quartile to cover data within another 1.5 times the interquartile range (the distance from the first to the third quartile). Any outliers are plotted individually.

We also collected data on increased memory requirements when running with MPFR. See Figure 37. Here, too, numbers are normalized with regard to the memory requirements of Surfer in standard double-precision floating-point mode. As expected, memory utilization goes up linearly with increased precision: at an MPFR precision of 100 bits the memory requirement is about 4 times that of Surfer's standard mode of operation; at 1000 bits we observed a factor of approximately 10 and at 4000 bits that factor is 30.

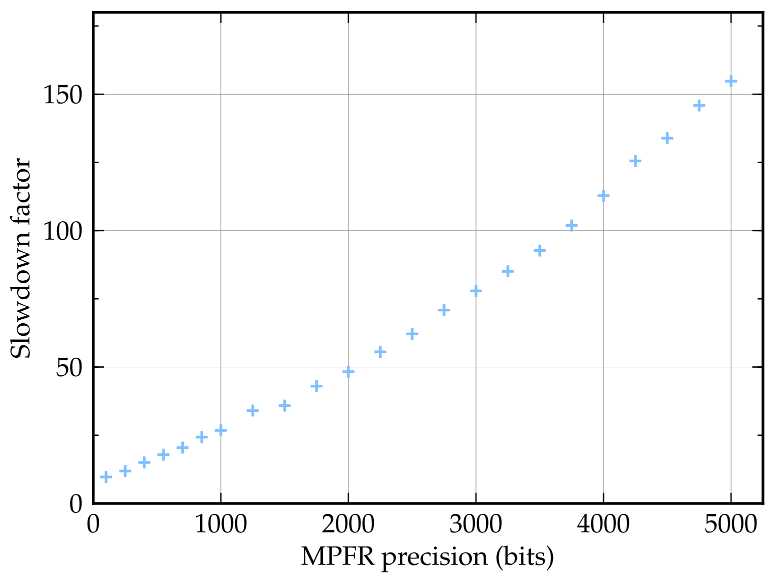


Figure 36: Slowdown when using Surfer's MPFR backend compared to its IEEE 754 double-precision backend.

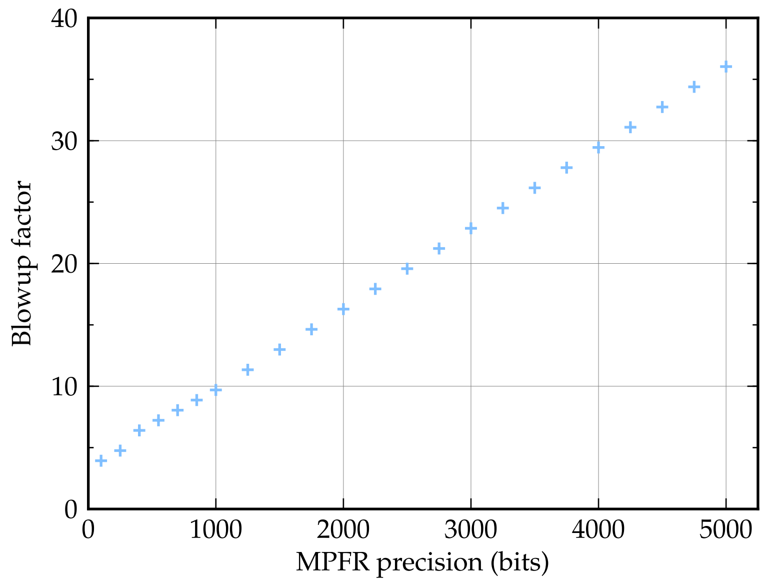


Figure 37: Memory use increase when using Surfer's MPFR backend compared to its IEEE 754 double-precision backend.

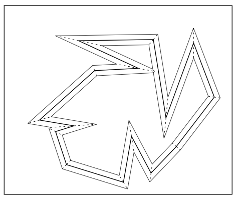
7 CONCLUSION

We studied Aichholzer and Aurenhammer's algorithm to construct the straight skeletons of planar straight-line graphs in detail.

We highlighted shortcomings of the original algorithm when input is not in general position and we presented solutions that work both with arbitrary and with finite precision arithmetic.

We implemented Aichholzer and Aurenhammer's algorithm and our extensions. Furthermore, we performed extensive tests on our implementation. We presented strong experimental evidence that the number of flip events, which are bound by $\mathcal{O}(n^3)$ in theory, are linear in practice.

Our code runs in $\mathcal{O}(n \log n)$ time in practice and $\mathcal{O}(n)$ space on all tested inputs. It clearly is the fastest straight skeleton code we are aware of.



A GALLERY

In the following figures the input graph is always shown in black while the straight skeleton is drawn in blue. Some input graphs were provided by Martin Held.

A.1 POLYGONS

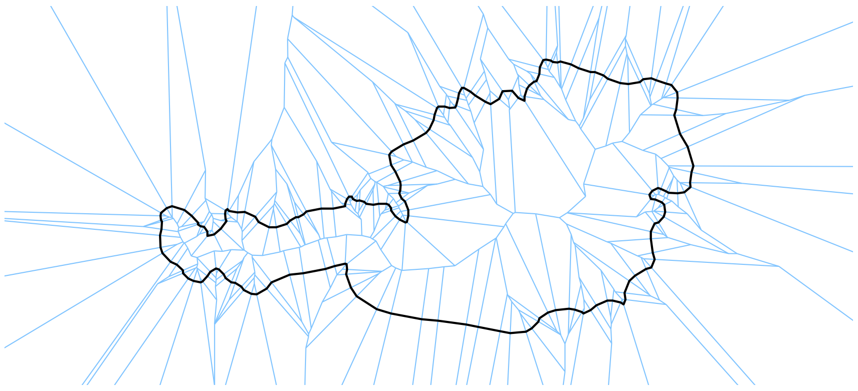


Figure 38: Borders of Austria.

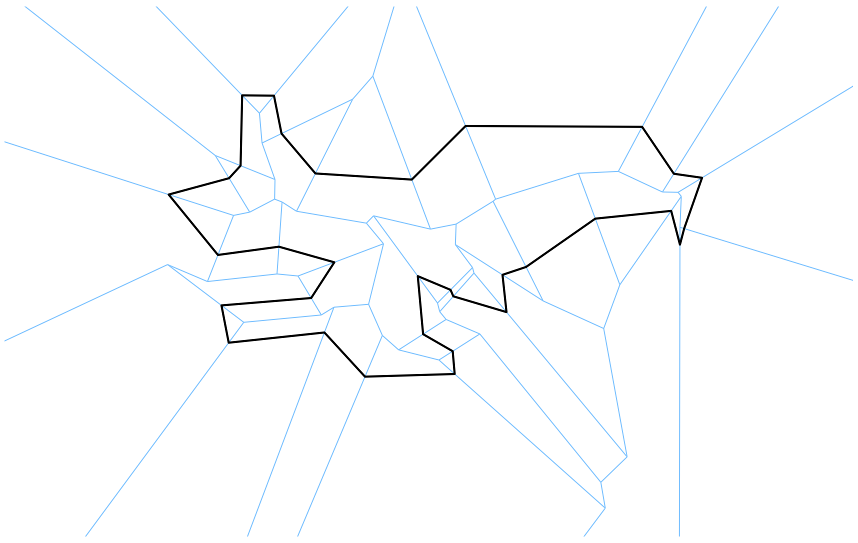
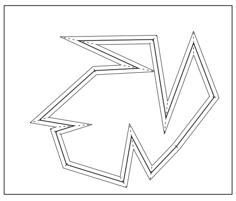


Figure 39: A random polygon.



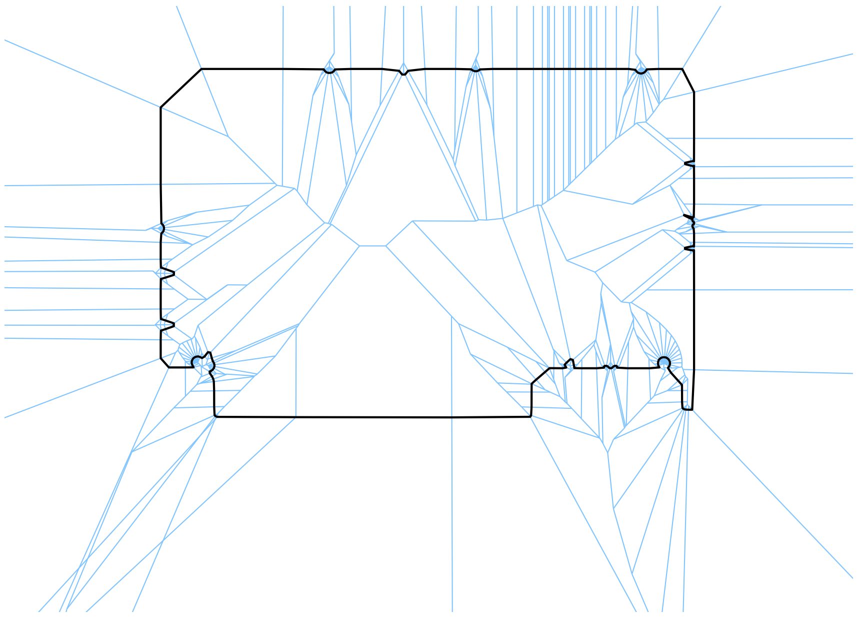


Figure 40: A printed circuit board outline.

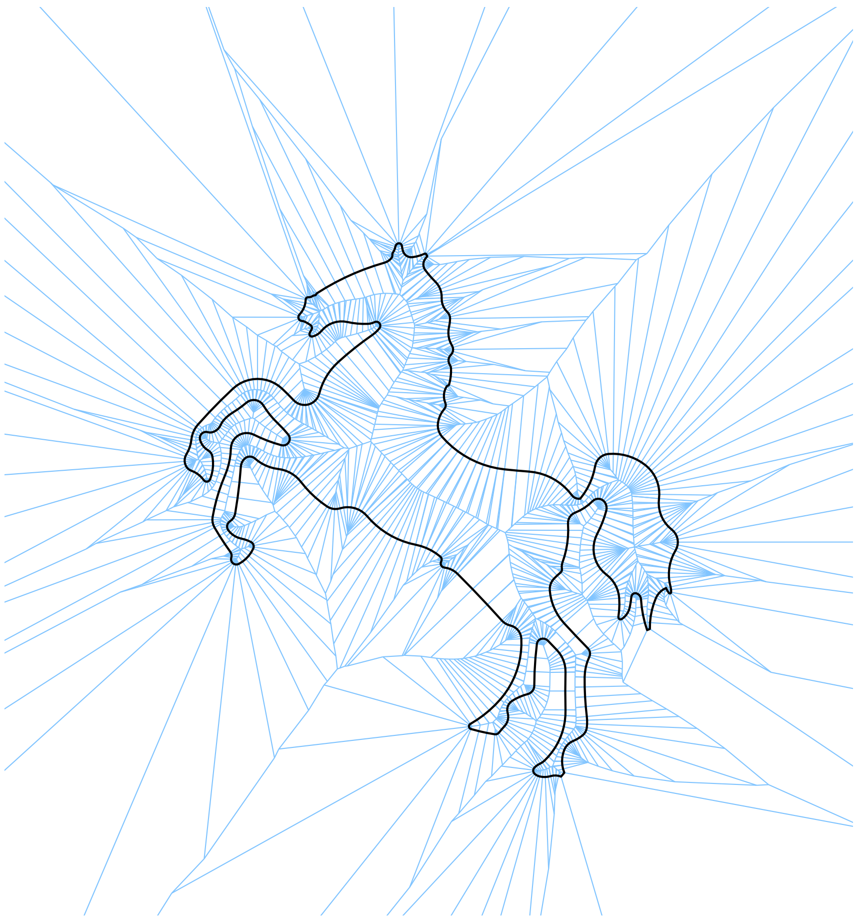


Figure 41: A Horse.

A.2 POLYGONS WITH HOLES

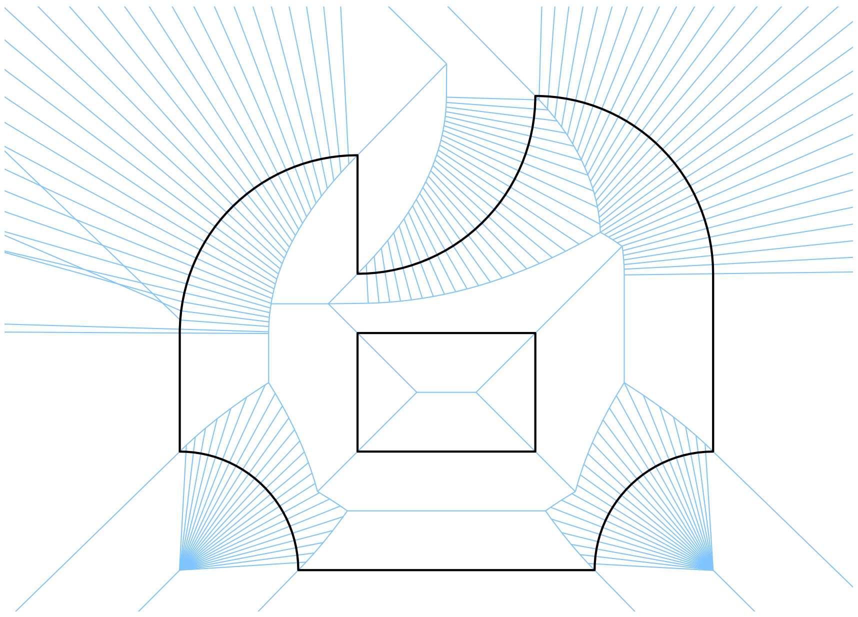


Figure 42: A polygon with one hole. Sampled arcs induce these sunshine-ray like patterns in the straight skeleton.

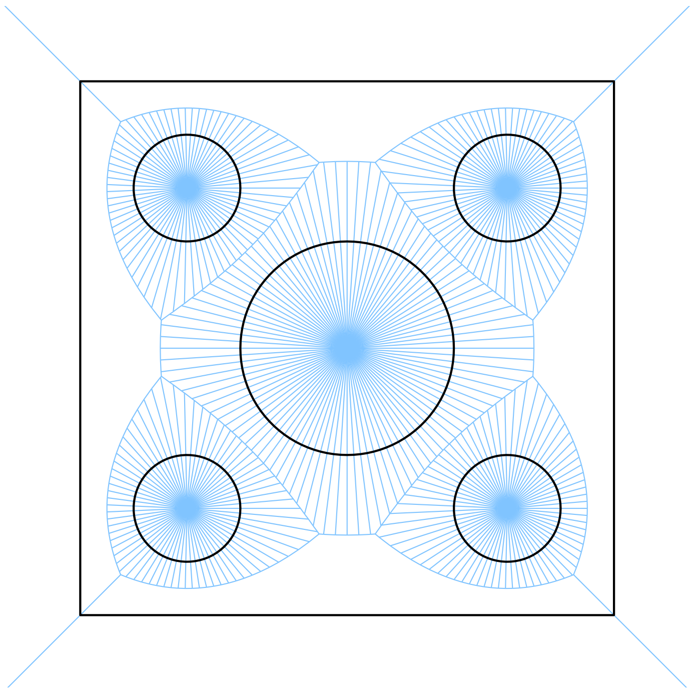
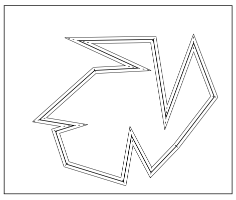


Figure 43: Several sampled circular holes in a square.



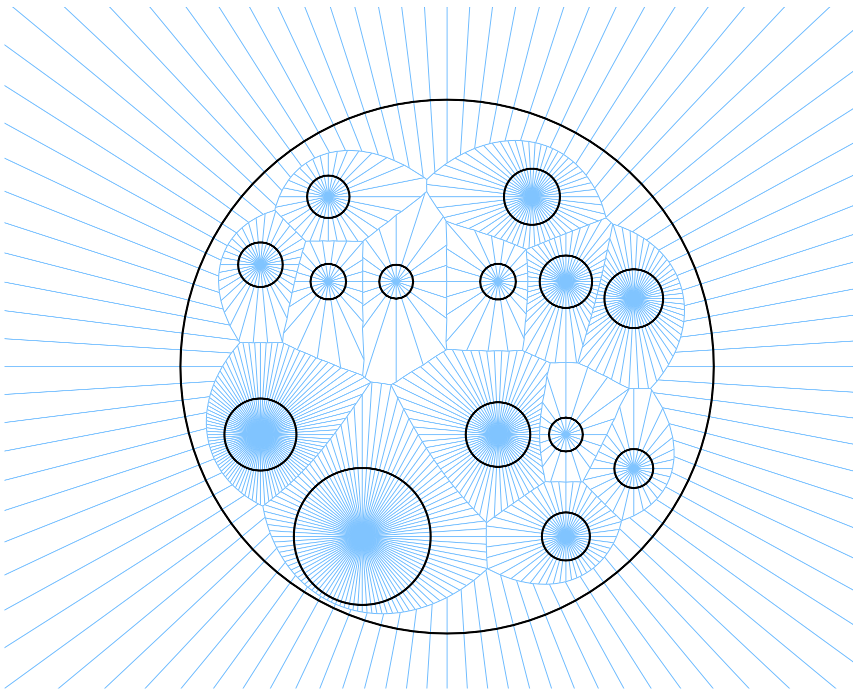


Figure 44: More circular holes.

A.3 OTHER STRAIGHT-LINE GRAPHS

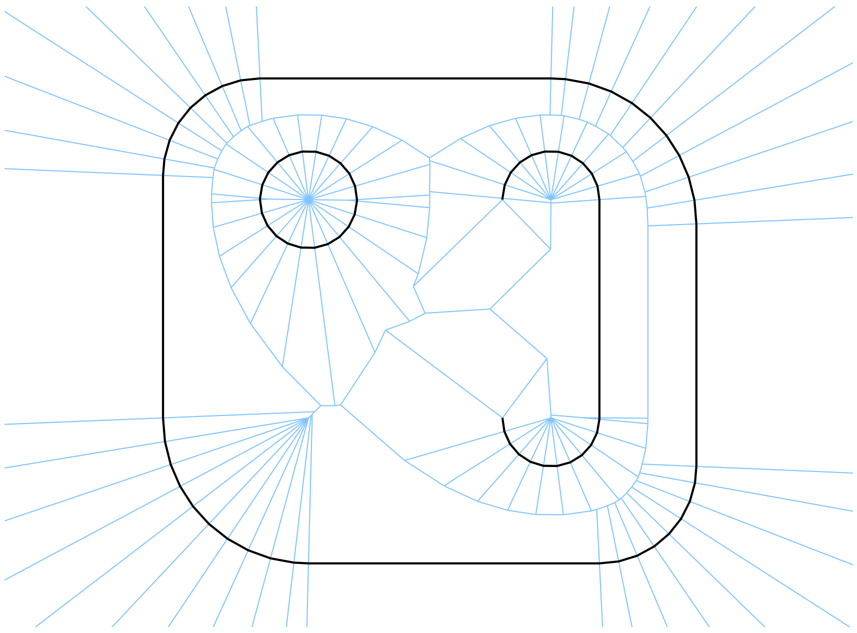


Figure 45: Almost a polygon with two holes – note the missing vertical input edge.

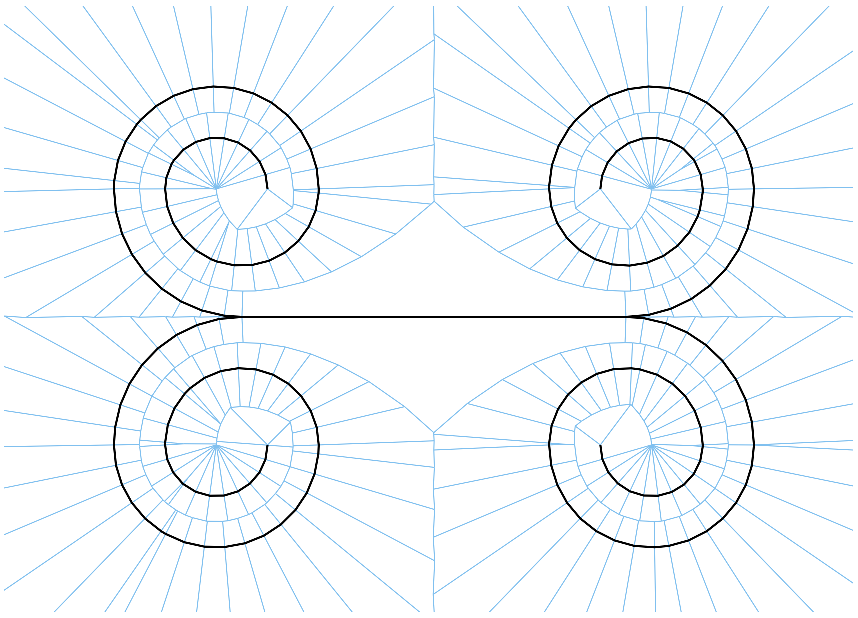


Figure 46: A set of spirals. The more densely sampled versions of these data sets cause our implementation to spend about 70 % to 80 % of its time in the triangulation phase and only very little in the wavefront propagation process — see Section 6.4.

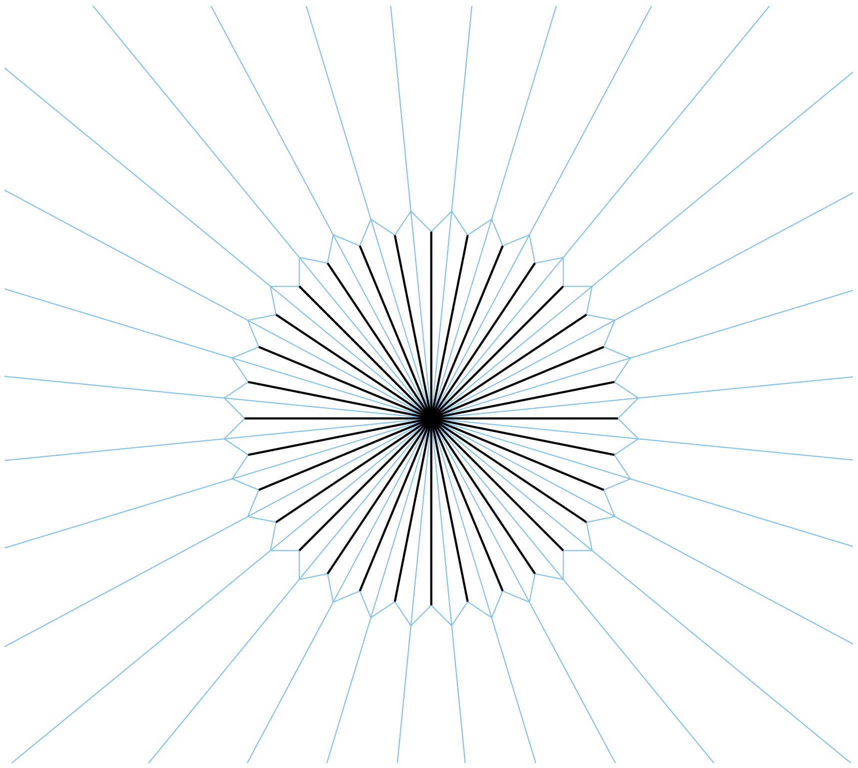
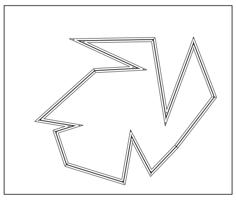


Figure 47: Stars are a second kind of input type which cause more work during triangulation than in actually computing the straight skeleton.



A.4 PROPAGATION PROCESS

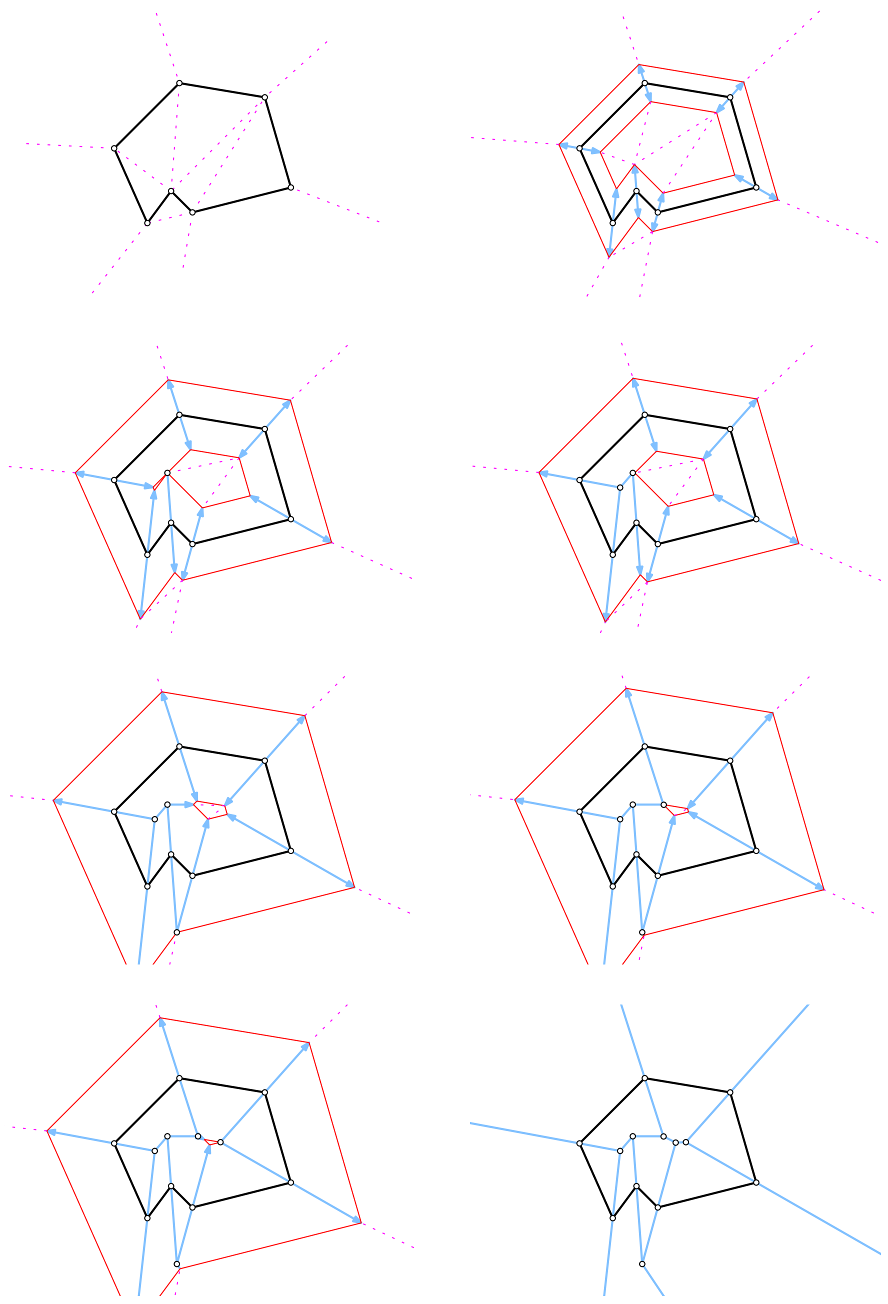


Figure 48: Wavefront propagation while computing the straight skeleton: In the first figure (top left) we show the input graph with a triangulation. The second figure (top right) shows the wavefront in red after propagation has started and the traces of wavefront vertices in blue. The following figures show the wavefront and triangulation at the collapse times of triangles, that is, whenever we have handled an event and a wavefront vertex has been stopped. The last figure shows the final straight skeleton.

B

BIBLIOGRAPHY

[AA96] Oswin Aichholzer and Franz Aurenhammer. Straight Skeletons for General Polygonal Figures in the Plane. In Jin-Yi Cai and C.K. Wong, editors, *COCOON'96 – 2nd Annual International Computing and Combinatorics Conference*, volume 1090 of *Lecture Notes in Computer Science*, pages 117–126. Springer-Verlag, June 1996.

[AA98] Oswin Aichholzer and Franz Aurenhammer. Straight Skeletons for General Polygonal Figures in the Plane. *Voronoi's Impact on Modern Sciences II*, pages 7–21, 1998.

[AAAG95a] Oswin Aichholzer, David Alberts, Franz Aurenhammer, and Bernd Gärtner. Straight Skeletons of Simple Polygons. *Proceedings of the 4th International Symposium of LIESMARS*, pages 114–124, October 1995.

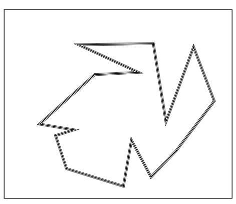
[AAAG95b] Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. A Novel Type of Skeleton for Polygons. *Journal of Universal Computer Science*, 1(12):752–761, 1995.

[ADD+13] Zachary Abel, Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Anna Lubiw, André Schulz, Diane Souvaine, Giovanni Viglietta, and Andrew Winslow. Algorithms for Designing Pop-Up Cards. In *Proceedings of the 30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, pages 269–280, Kiel, Germany, February 27–March 2 2013.

[AH96] Thomas Auer and Martin Held. Heuristics for the Generation of Random Polygons. In *CCCG 1996 – 8th Canadian Conference on Computational Geometry. Proceedings*, pages 38–44, Ottawa, Canada, August 1996. Carleton University Press.

[AM94] Pankaj K. Agarwal and Jirí Matousek. On Range Searching with Semialgebraic Sets. *Discrete & Computational Geometry*, 11:393–418, 1994.

[Cac12] Fernando Cacciola. 2D Straight Skeleton and Polygon Offsetting. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012.



[CGA] Computational Geometry Algorithms Library. http://www.cgal.org/.

[Cha91] Bernard Chazelle. Triangulating a Simple Polygon in Linear Time. *Discrete & Computational Geometry*, 6:485–524, 1991.

[Cha93] Bernard Chazelle. Cutting Hyperplanes for Divide-and-Conquer. *Discrete & Computational Geometry*, 9(2):145–158, 1993.

[Che] Otfried Cheong. The Ipe extensible drawing editor. http://ipe7.sourceforge.net/.

[CV02] Siu-Wing Cheng and Antoine Vigneron. Motorcycle Graphs and Straight Skeletons. In *SODA 2002 – 13th Annual ACM-SIAM Symposium on Discrete Algorithms. Proceedings*, page 156–165, San Francisco, CA, USA, January 2002.

[DDL98] Erik D. Demaine, Martin L. Demaine, and Anna Lubiw. Folding and Cutting Paper. In *Revised Papers from the Japan Conference on Discrete and Computational Geometry (JCDCG'98)*, volume 1763 of *Lecture Notes in Computer Science*, pages 104–117, Tokyo, Japan, December 9–12 1998.

[DDM00] Erik D. Demaine, Martin L. Demaine, and Joseph S. B. Mitchell. Folding Flat Silhouettes and Wrapping Polyhedral Packages: New Results in Computational Origami. *Computational Geometry: Theory and Applications*, 16(1):3–21, 2000. Special issue of selected papers from the 3rd CGC Workshop on Computational Geometry, 1998.

[DO07] Erik D. Demaine and Joseph O'Rourke. *Geometric Folding Algorithms: Linkages, Origami, Polyhedra*. Cambridge University Press, July 2007.

[EE99] David Eppstein and Jeff Erickson. Raising Roofs, Crashing Cycles, and Playing Pool: Applications of a Data Structure for Finding Pairwise Interactions. *Discrete & Computational Geometry*, 22(4):569–592, 1999.

[FHL$^+$07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007. http://www.mpfr.org/.

[HH10a] Stefan Huber and Martin Held. Computing Straight Skeletons of Planar Straight-Line Graphs Based on Motorcycle Graphs. In *CCCG 2010 – 22nd Canadian Conference on Computational Geometry. Proceedings*, pages 187–190, Winnipeg, Canada, August 2010.

[HH10b] Stefan Huber and Martin Held. Straight Skeletons and their Relation to Triangulations. In *EuroCG '10 – 26th European Workshop on Computational Geometry. Proceedings*, pages 189–192, Dortmund, Germany, March 2010.

[HH11] Stefan Huber and Martin Held. Theoretical and Practical Results on Straight Skeletons of Planar Straight-Line Graphs. In *SoCG '11 – 27th Symposium on Computational Geometry 2011. Proceedings*, pages 171–178, Paris, France, June 2011.

[Hoh] Markus Hohenwarter. GeoGebra – Dynamic mathematics & science for learning and teaching. http://www.geogebra.org/.

[HOS96] Sabine Hanke, Thomas Ottmann, and Sven Schuierer. The Edge-Flipping Distance of Triangulations. *Journal of Universal Computer Science*, 2:570–579, 1996.

[Hub12] Stefan Huber. *Computing Straight Skeletons and Motorcycle Graphs: Theory and Practice*. Shaker Verlag, April 2012. ISBN 978-3-8440-0938-5.

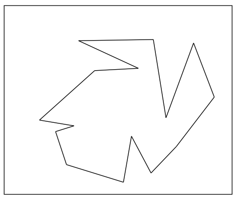
[LD03] Robert G. Laycock and Andrew M. Day. Automatically Generating Large Urban Environments based on the Footprint Data of Buildings. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, SM '03, pages 346–351, New York, NY, USA, 2003. ACM.

[MPF] The GNU MPFR Library. http://www.mpfr.org/.

[OGL] OpenGL. https://www.opengl.org/.

[PHH12] Peter Palfrader, Martin Held, and Stefan Huber. On Computing Straight Skeletons by Means of Kinetic Triangulations. In *Algorithms – ESA 2012 – 20th Annual European Symposium. Proceedings*, pages 766–777, Ljubljana, Slovenia, September 2012.

[She96] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*,



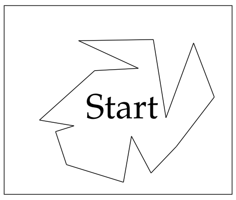
pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry. https://www.cs.cmu.edu/~quake/triangle.html.

[Sug13] Kokichi Sugihara. Design of Pop-Up Cards Based on Weighted Straight Skeletons. In *ISVD 2013 – 10th International Symposium on Voronoi Diagrams in Science and Engineering. Proceedings*, pages 23–28, Saint Petersburg, Russia, July 2013.

[TS12] Akiyasu Tomoeda and Kokichi Sugihara. Computational Creation of a New Illusionary Solid Sign. In *ISVD 2012 – 9th International Symposium on Voronoi Diagrams in Science and Engineering. Proceedings*, ISVD '12, pages 144–147. IEEE Computer Society, 2012.

C

LIST OF FIGURES



Except where otherwise noted, all figures in this thesis were created by the author. The creation process involved only their own software or third-party software used with permission. No figures infringe any rights of third parties. In particular, straight skeletons were created with either Bone by Stefan Huber or with Surfer by the author, or were constructed by hand by the author.