On Computing Straight Skeletons by Means of Kinetic Triangulations*

Peter Palfrader[1], Martin Held[1], and Stefan Huber[2]

[1] FB Computerwissenschaften, Universität Salzburg, A-5020 Salzburg, Austria
[2] FB Mathematik, Universität Salzburg, A-5020 Salzburg, Austria
{ppalfrad,held,shuber}@cosy.sbg.ac.at

Abstract. We study the computation of the straight skeleton of a planar straight-line graph (PSLG) by means of the triangulation-based wavefront propagation proposed by Aichholzer and Aurenhammer in 1998, and provide both theoretical and practical insights. As our main theoretical contribution we explain the algorithmic extensions and modifications of their algorithm necessary for computing the straight skeleton of a general PSLG within the entire plane, without relying on an implicit assumption of general position of the input, and when using a finite-precision arithmetic. We implemented this extended algorithm in C and report on extensive experiments. Our main practical contribution is (1) strong experimental evidence that the number of flip events that occur in the kinetic triangulation of real-world data is linear in the number n of input vertices, (2) that our implementation, Surfer, runs in $\mathcal{O}(n \log n)$ time on average, and (3) that it clearly is the fastest straight-skeleton code currently available.

1 Introduction

1.1 Motivation

The straight skeleton of a simple polygon is a skeletal structure similar to the generalized Voronoi diagram, but comprises straight-line segments only. It was introduced to computational geometry by Aichholzer et al. [1], and later generalized to planar straight-line graphs (PSLGs) by Aichholzer and Aurenhammer [2]. Currently, the most efficient straight-skeleton algorithm for PSLGs, by Eppstein and Erickson [7], has a worst-case time and space complexity of $\mathcal{O}(n^{17/11+\epsilon})$ for any $\epsilon > 0$. For a certain class of simple polygons with holes Cheng and Vigneron [6] presented a randomized algorithm that runs in $\mathcal{O}(n\sqrt{n}\log^2 n)$ time. However, both algorithms employ elaborate data structures in order to achieve these complexities and are not suitable for implementation.

The first comprehensive straight-skeleton code was implemented by Cacciola [4] and is shipped with the CGAL library [5]. It handles polygons with holes as input, and requires $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n^2)$ space for real-world datasets,

---

see [10] for an explanation and experimental analysis. The code Bone by Huber and Held [10] has been, until now, the fastest implementation. It handles PSLGs as input and runs in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space in practice.

The algorithm by Aichholzer and Aurenhammer [2] propagates a wavefront by means of kinetic triangulations and it can handle PSLGs as input. No better complexity bound than $\mathcal{O}(n^3 \log n)$ has been established so far, due to only a trivial $\mathcal{O}(n^3)$ upper bound on the number of flip events that might occur in the triangulation in the worst case, even though no input is known that requires more than $\mathcal{O}(n^2 \log n)$ time. The basic algorithm is suitable for implementation, but no full implementation has been published. In fact, if general position cannot be assumed then one needs to fill in algorithmic gaps prior to an actual implementation. For instance, the basic algorithm of [2] may loop if multiple concurrent events are not processed properly, even if exact arithmetic is used.

Our contribution. We thoroughly investigate Aichholzer and Aurenhammer's triangulation-based approach [2] both from a theoretical and a practical point of view. We start with a discussion of our extensions to their basic algorithm, which are necessary in order to handle arbitrary PSLGs. In particular, we present a procedure for resolving a loop of events in the kinetic triangulation, we report on technical details concerning unbounded triangles in order to triangulate the entire plane, and we discuss how to handle parallel edges in G that lead to infinitely fast vertices in the kinetic triangulation. Besides these extensions, we address a major open question raised by Aichholzer and Aurenhammer [2]: How many flip events shall we expect to occur in the kinetic triangulation of practical data?

We implemented the basic algorithm and our extensions in C, and present extensive statistics based on test runs for about 20 000 industrial and synthetic datasets of different characteristics. As first important practical contribution we provide strong experimental evidence for Aichholzer and Aurenhammer's conjecture that only $\mathcal{O}(n)$ flip events suffice for all practical datasets.

Our code, Surfer, can be run with two arithmetic back-ends: (i) with double-precision floating-point arithmetic, and (ii) with the extended-precision library MPFR [8]. Tests clearly show that Surfer runs in $\mathcal{O}(n \log n)$ time for all our datasets. Furthermore, it is reliable enough to handle datasets of a few million vertices with floating-point arithmetic. In comparison with the code provided by CGAL, Surfer has several advantages: (i) it is by a linear factor faster in practice, (ii) it can handle PSLGs as input, and (iii) its space complexity is in $\mathcal{O}(n)$. Furthermore, even though the worst-case complexity of Surfer is worse than that of Bone [10], our code turns out to be faster by a factor of 10 in runtime tests.

1.2 Preliminaries and Basic Definitions

Definition of the straight skeleton. Aichholzer et al. [1] defined a straight skeleton $\mathcal{S}(G)$ of a PSLG G based on a so-called wavefront propagation process. The idea is that every edge of G sends out two wavefront copies that move in a

parallel fashion and with constant speed on either side, see Fig. 1. Consider a non-terminal vertex v of G: As the wavefront progresses, wavefront vertices, i.e., copies of v, move along angular bisectors defined by pairs of consecutive edges (in the cyclic incidence order) that are incident at v. At a terminal vertex v of G an additional wavefront edge orthogonal to the incident input edge is emanated such that the wavefront forms a rectangular cap at v. (It is assumed that G contains no isolated vertices.)

We denote by $\mathcal{W}_G(t)$ the wavefront at time t and interpret $\mathcal{W}_G(t)$ as a 2-regular graph which has the shape of a mitered offset curve of G. During the propagation of $\mathcal{W}_G$ topological changes occur:

- An *edge event* occurs when a wavefront edge shrinks to zero length and vanishes.
- A *split event* occurs when a reflex wavefront vertex meets a wavefront edge and splits the wavefront into parts. We call a wavefront vertex *reflex* if the angle of the incident wavefront edges on the propagation side is reflex.
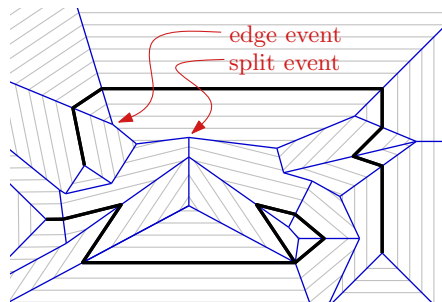
Fig. 1. A PSLG G, wavefronts $\mathcal{W}_G(t)$ in gray for different times t, and the straight skeleton $\mathcal{S}(G)$.

The straight skeleton $\mathcal{S}(G)$ is then defined as the union of loci that are traced out by the wavefront vertices, see Fig. 1. Every vertex of $\mathcal{S}(G)$ is due to an event.

The triangulation-based algorithm. The idea that drives the triangulation-based algorithm by Aichholzer and Aurenhammer [2] is to keep for any $t \geq 0$ the area $\mathbb{R}^2 \setminus \bigcup_{t' < t} \mathcal{W}_G(t')$ triangulated. In other words, they maintain a kinetic triangulation of those parts of the plane that have not yet been swept by the wavefront $\mathcal{W}_G$. Every edge event and every split event is indicated by the collapse of a triangle as either (i) a wavefront edge collapsed to zero length or (ii) a wavefront vertex met a wavefront edge. Hence, the topological changes of the wavefront are indicated by the topological changes of the kinetic triangulation. However, some topological changes of the kinetic triangulation do not correspond to a wavefront event, namely when a reflex wavefront vertex meets an inner triangulation edge. One needs to flip this edge in order to maintain a valid triangulation. Hence, these events are called *flip events*. We follow the notation of Aichholzer and Aurenhammer and call an inner triangulation diagonal a *spoke* in the remainder of this paper.

Their algorithm starts with a constrained triangulation of G and adapts it to a triangulation for the initial wavefront $\mathcal{W}_G(0)$ by duplicating the edges of G and inserting zero-length edges that are emanated at terminal vertices of G. For every triangle a collapse time is computed. If a triangle collapses in finite time it is put into a priority queue Q prioritized by its collapse time. Then one event after the other is fetched from Q in chronological order, and the necessary topological

changes are applied to the kinetic triangulation. Eventually, Q is empty as no further triangle is collapsing in finite time. At that point all straight-skeleton nodes were computed.

Note that there are $\mathcal{O}(n)$ edge and split events as $\mathcal{S}(G)$ is of linear size. An edge and split event locally changes the topology of the wavefront and adapts the velocities of some vertices. Consequently, the collapse times of all incident triangles need to be recomputed. Hence, a single edge or split event may require $\mathcal{O}(n \log n)$ time, which leads to $\mathcal{O}(n^2 \log n)$ as total time complexity for all edge and split events.

The signed area of a triangle can be expressed as quadratic polynomial in t and, hence, a single triangle can collapse at most at two single points in time. As there are at most $\binom{n}{3}$ combinatorial possibilities for triangles among n vertices over the entire simulation time, there are at most $\mathcal{O}(n^3)$ flip events. This is the best known bound. A single flip event requires $\mathcal{O}(1)$ modifications in Q and, thus, can be handled in $\mathcal{O}(\log n)$ time. In total, the algorithm has a worst-case complexity of $\mathcal{O}((n^2 + k) \log n) \subseteq \mathcal{O}(n^3 \log n)$, where $k \in \mathcal{O}(n^3)$ denotes the number of flip events. However, no input is known that causes more than $\mathcal{O}(n^2)$ flip events.

2   Handling Unbounded Triangles

If the entire straight skeleton of a general PSLG G is to be computed, one has to ensure that the initial triangulation covers a portion of the plane that is large enough to contain all nodes of $\mathcal{S}(G)$. The natural idea of computing $\mathcal{S}(G)$ inside of a "large" box or triangle is difficult to cast into a reliable implementation for two reasons: First, no efficient method is known for computing a good upper bound on the maximum distance of a node of $\mathcal{S}(G)$ from G. Second, picking a truly large box might be a heuristic attempt to ensure coverage for all practically relevant needs but it will result in lots of very skinny triangles. These triangles are difficult to process correctly on a finite-precision arithmetic, and they place a burden on floating-point filters used to speed up exact geometric computing.

Therefore we construct a triangulation of the entire plane as follows: First, we compute a constrained Delaunay triangulation of G inside of its convex hull, $CH(G)$. Then we attach an *unbounded triangle* to every edge of $CH(G)$. Such an unbounded triangle has one finite edge on $CH(G)$ and two unbounded edges. These unbounded edges are thought to meet at infinity.

While computing the collapse time of a (finite) triangle with three vertices moving at constant speed amounts to solving a quadratic equation in one variable, it is less obvious how to deal with unbounded triangles. We note that it is not sufficient to regard an unbounded triangle as collapsed only if its finite edge has shrunk to zero length. This would allow unbounded triangles to move to the interior of the wavefront, causing us to miss events that change the topology of the wavefront later on: The left part of Fig. 2 depicts a portion of a PSLG G (in bold) such that the input edges, or wavefronts, w_1, w_2, w_3 and the spokes s_1, s_2 lie on $CH(G)$. If the angles at the reflex wavefront vertices are chosen
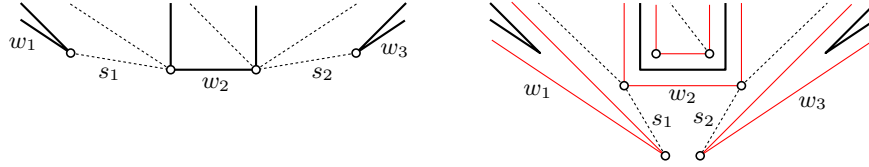
Fig. 2. A crash of wavefronts might be missed if unbounded triangles are handled naively.

appropriately then the wavefronts will collide while no triangle collapse serves as a witness of the event, see the right part.

In order to prevent such problems we proceed as follows: We consider the stereographic projection of the plane $\mathbb{R}^2$ to the sphere S^2, which maps the origin to the south pole of S^2 and the north pole represents all points at infinity. Every triangle of our triangulation, including the unbounded triangles, is mapped to a spherical triangle on S^2. The infinite edges of unbounded triangles are arcs of great circles supporting the north and south pole of S^2.

We now regard an unbounded triangle of $\mathbb{R}^2$ as collapsed when its spherical counterpart collapsed, i.e., when its three vertices lie on a great circle. A collapse of an unbounded triangle indicates either an edge event or a flip event, and both events can be handled in a similar fashion as for bounded triangles. Of course, it would be quite cumbersome for an actual implementation to compute collapse times of spherical triangles. Fortunately this can be avoided: The spherical counterpart of an unbounded triangle Δ collapses if the two finite vertices v_1, v_2 and the north pole lie on a great circle. This is the case if and only if v_1, v_2 and the south pole lie on a great circle. Hence Δ collapses if the two finite vertices and the origin are collinear in the plane, i.e., if the triangle formed by v_1, v_2 and the origin collapses.

3 Handling Input without General Position Assumed

In this section we describe the nuts and bolts required for turning the basic algorithm into an implementation that can handle real-world data while using standard floating-point arithmetic. Problems arise because (1) the (usually implicit) assumption of general position (GPA) is not warranted for real-world applications, and (2) finite-precision arithmetic does not guarantee to process all events in the correct order. Note that working with finite precision arithmetic precludes approaches like symbolic perturbation.

3.1 Vertices Moving at Infinite Speed

Consider the c-shaped PSLG shown in Fig. 3(a) together with a wavefront and a part of the triangulation. As the wavefront progresses, the shaded triangle Δ_1 will collapse since the edge e between vertices v_1 and v_2 will shrink to zero
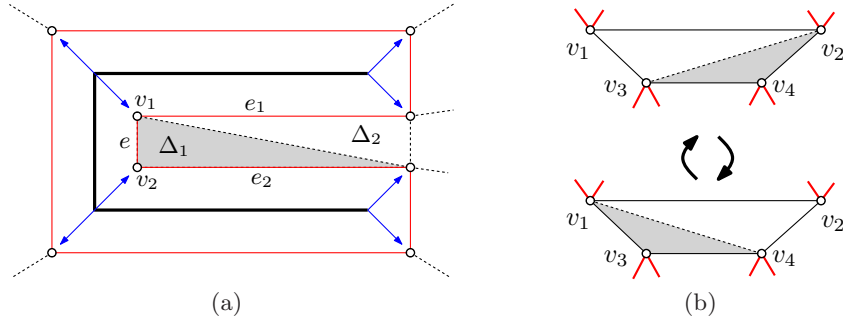
Fig. 3. Without general position assumed. (a) The edge event for the shaded triangle Δ_1 will create a vertex moving at infinite speed. (b) A loop of concurrent flip events.

length. (Triangle Δ_2 and some other triangles will collapse at the same time, but this is irrelevant at the moment.)

The standard procedure for an edge event is to replace the two vertices v_1, v_2 with a new vertex that moves along the angular bisector of the two incident wavefront edges, at the speed required to follow the wavefront propagation. In the situation shown in Fig. 3(a), the two wavefront edges e_1, e_2 are parallel and overlap at the time of the event. This means that the new vertex just created moves along the supporting line of e_1, e_2, but it has to move at infinite speed to "keep up" with the wavefront propagation since it runs perpendicular to the direction of the wavefront propagation.

This problem is resolved by introducing triangulation vertices that are marked as *infinitely fast*. Like any other vertex, such a vertex v is the central vertex of a triangle fan of one or more triangles, $\Delta_1, \Delta_2, \ldots, \Delta_n$. This fan is enclosed at v by two overlapping wavefronts. (Otherwise v would not be infinitely fast.) This implies that all triangles of the fan are collapsing at the time the infinitely fast moving vertex comes into existence. Among all these triangles we choose either Δ_1 or Δ_n, depending on which has the shorter wavefront edge to v. Let v' be the vertex next to v on the shorter wavefront edge.

In the chosen triangle 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.

3.2 Infinite Loops of Flip Events

At first sight, it appears evident that the basic triangulation-based algorithm terminates as there are only finitely many events to be processed. For edge and split events an even simpler argument can be applied: Every edge and split event reduces the number of triangles and, thus, only $O(n)$ many edge and split events can occur. While flip events do not result in a reduction of the triangle count,

we still make progress in the wavefront propagation if no two flip events occur at the same time.

However, if general position is not assumed and, thus, two or more (flip) events may occur at the same time then this standard argument for the termination of the basic triangulation-based algorithm fails. Fig. 3(b) shows two wavefront vertices v_1, v_2 that move downwards and two wavefront vertices v_3, v_4 that move upwards. Now assume that all four vertices will become collinear at some future point in time. Then the two triangles shown will collapse at the same time. Hence, we have two choices on how to proceed: We can either flip the spoke (v_1, v_2) or the spoke (v_2, v_3). If we chose to flip (v_1, v_2) and subsequently (v_2, v_3), then we would achieve progress as all four vertices could proceed with their movement. If, however, we chose to flip (v_2, v_3) then no progress would be achieved, and a subsequent flip of (v_1, v_4) would get us into an infinite loop. Modifying the set-up of Fig. 3(b) by regarding (v_1, v_2) as an edge of the wavefront yields an example for a possible event loop that involves a split event and a flip event.

This simple example can be made more complex in order to incorporate multiple concurrent split and flip events. We emphasize that such a loop of events can occur even if exact arithmetic is used for computing all event times without numerical error. That is, this is a genuine algorithmic problem once we allow inputs that trigger multiple events at the same time.

Indications for a flip event loop. One might suspect that processing an event twice is a clear indication that we encountered the same triangulation a second time and that the event processing ended up in a loop. However, this is not correct, as demonstrated in Fig. 4. Assume that the vertices v_1 and v_6 move downwards while the vertices $v_2, \ldots, v_5$ move upwards such that all six vertices become collinear at some point t in time, lined up in the order $v_1, \ldots, v_6$, with no two vertices coinciding. Hence, all four triangles depicted have collapse events scheduled for time t. In Fig. 4, the collapse event chosen is indicated by shading, and the next triangulation is obtained by flipping the dashed spoke. The triangle Δ_1 is processed twice and still all eight triangulations are different and, thus, we did not end up in a loop. However, if in (viii) we continue with Δ_3 instead of, say, Δ_2, then we obtain the same triangulation as in (v) and we are indeed caught in a loop.

In general, it is a save to handle non-flip events prior to flip events, as non-flip events reduce the number of triangles. However, preferring concurrent non-flip events over flip events requires that one notice the event times are identical, which is prone to errors when using finite-precision arithmetic. And, of course, we were still left with the problem of detecting and handling event loops that consist only of concurrent flip events. Summarizing, two questions need to be addressed: (i) how to detect event loops and (ii) how to cope with them. We pay particular attention to handling event loops on finite-precision arithmetic, since our implementation operates with double-precision or MPFR-based extended precision arithmetic.
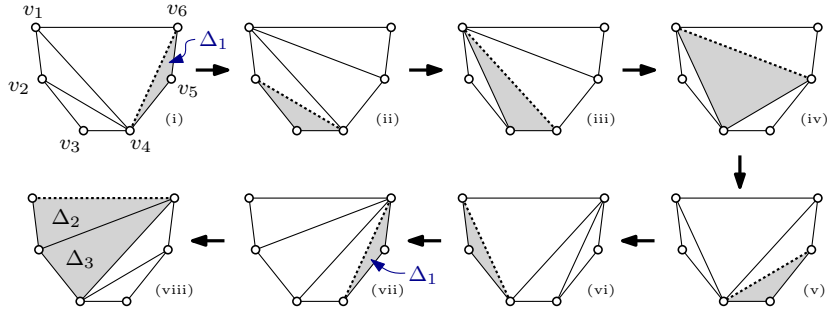
Fig. 4. Encountering an event twice need not imply a loop nor the same triangulation.

Handling flip event loops. In order to detect and handle event loops we maintain a history H that saves for every processed event a triple (t^*, t, Δ), where t^* denotes the number of triangles remaining in the triangulation, Δ denotes the sorted vertex-triple of the collapsed triangle, and t denotes its collapse time. These triples are stored in the same sequential order as the corresponding events are processed. In addition, we maintain a search data structure S that stores the triples (t^*, t, Δ) in lexicographical order. Note that every non-flip event decrements t^*. Thus, t^* measures the progress of the wavefront propagation in a discrete manner, and we can empty both data structures whenever t^* is decreased.

Whenever an event is to be processed, we check whether the corresponding triple is already stored in S. If we ended up in a loop then we are guaranteed to find that triple already stored in S. As explained in Fig. 4, the opposite conclusion need not be true. Nonetheless, once we observe that the triple (t^*, t, Δ) was already handled, we apply the following method in order to resolve a potential loop. (No harm is caused in case of a false alarm.)

First of all, as t^* has not changed, the entire potential loop comprises flip events only. With exact arithmetic, all triples between the first occurrence T_1 and second occurrence T_2 of (t^*, t, Δ) in H have the identical value for t. With finite-precision arithmetic, we declare all triples between T_1 and T_2 to have happened at the same time, even though there may be slight deviations in time for the triples between T_1 and T_2.

Next, we trace back the triples in H from T_2 to T_1 and mark all triangles that are known to have collapsed due to their role in a flip event. An inductive argument shows that the union of these triangles forms one or more polygons that have collapsed to straight-line segments. Let P denote the one polygon that contains Δ. We now roll back all triples between T_2 and T_1 that flip edges in P, including T_2. That is, we start at T_2 and visit all triples in H until we reach T_1, and if the corresponding triangle is in P we undo the flip and remove the triple from H and S. (Recall that only flips occurred between T_1 and T_2.)

We now adapt the triangulation for P and for the triangles Δ_e opposite to edges e of P as follows, see Fig. 5. Let $v_1, \ldots, v_k$ denote the vertices of P in sorted order with respect to the line to which P collapsed. First, we ensure that
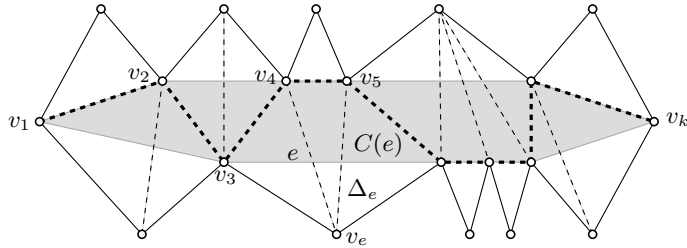
Fig. 5. An entire polygon P (shaded) collapsed to a line. We adapt the triangulation within P and the triangles Δ_e attached to edges e of P by first inserting a monotone chain $v_1, \ldots, v_k$ (bold dashed) and subsequently triangulating the resulting cells of P and the triangles Δ_e.

$v_i v_{i+1}$ is a diagonal for all $1 \le i < k$. That is, the path $v_1, \ldots, v_k$ is part of the triangulation. This path tessellates P into cells $C(e)$ that are bounded by two or more edges of the path and a single edge e of P. We denote by v_e the opposite vertex of e in Δ_e, see Fig. 5. Then we triangulate for each cell $C(e)$ the area $C(e) \cup \Delta_e$ such that every diagonal is incident to v_e.

Let T^* denote the last triple in H after the rollback. We use an algorithm by Hanke et al. [9] to append after T^* a sequence of triples that transfers the original triangulation (at the state just after T^*) into the new triangulation explained above. Similarly, those triples are inserted into S. Each of the new triples of H is also furnished with a pointer that points to T_1. The idea behind this pointer is that the time values of all triples between T_1 and triples that point to T_1 are considered to be equal. (With exact arithmetic they all are indeed equal.) After this reconfiguration of the triangulation and of H and S we proceed as usual.

Assume now that one of the triangles Δ_e has zero area as v_e was collinear with the vertices of P, too. If our loop-detection method later reports another potential loop because a triple T_3, which is in H, would be processed twice and T_3 contains a pointer back to T_1 then we repeat our resolution method presented above. However, the only difference is now that we consider all events between T_1 and T_3 to happen concurrently. That is, the rollback phase does not stop at T_3 but is extended back to T_1. As a consequence, the resulting new polygon includes the old polygon P, and we detected even more vertices that are collinear at that particular time. As there are only finitely many triangles, at some point in time, we obtain a largest polygon P. Hence, it is guaranteed that the above method also terminates on finite-precision arithmetic.

Of course, we make sure to compute all numerical values used by our algorithm in a canonical way. In particular, different computations of the collapse time of the same triangle are guaranteed to yield precisely the same numerical value. Also, note that we carefully avoid the use of precision thresholds for judging whether the collapse times of two triangles are identical: On a finite-precision arithmetic this would be prone to errors and one could not guarantee that the algorithm will terminate.

Sorting vertices along the collapse line. Finally, we discuss a subtle detail when sorting the vertices $v_1, \ldots, v_k$ along the line to which P collapsed. First, we determine the minimum $t_{\min}$ and the maximum $t_{\max}$ among the collapse times of the triangles in P. Next, we determine a fitting straight line $L_{\min}$ ($L_{\max}$, resp.) of the vertices $v_1, \ldots, v_k$ at time $t_{\min}$ ($t_{\max}$, resp.) by means of a least-square fitting. Then we sort $v_1, \ldots, v_k$ at time $t_{\min}$ with respect to $L_{\min}$ and obtain the sequence $v_{i_1}, \ldots, v_{i_k}$; likewise for $t_{\max}$ and $L_{\max}$. If we obtain the same order then we proceed with it as described above. If, however, a vertex v_{i_j} has a different position j' in the sorted sequence with respect to $t_{\max}$ then we declare the vertices $v_{i_{j'}}$ and v_{i_j} to coincide. Furthermore, we enforce a spoke e between those two vertices in the triangulation and handle one of both non-flip events that correspond to the collapse of the two triangles sharing e. Consequently, t^* is decremented and we again have a guaranteed progress of our algorithm.

4 Experimental Results

We implemented the full wavefront-propagation algorithm in C. The resulting code, Surfer, can be run with two arithmetic back-ends: (i) with double-precision floating-point arithmetic, and (ii) with the arbitrary-precision library MPFR [8].

We tested Surfer on about twenty thousand polygons and PSLGs, with up to 2.5 million vertices per data. 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 [3], 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.
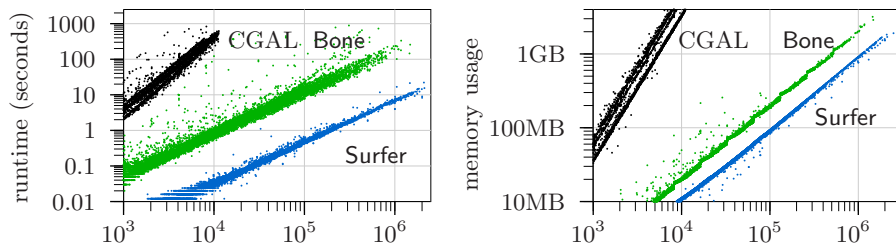
4.1 Number of Flip Events
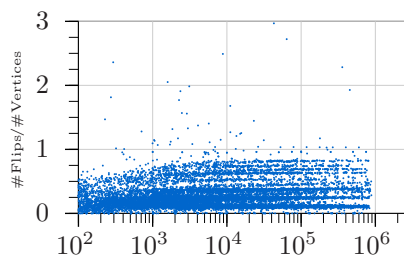
Fig. 6. The number of flip events is linear in practice. (input size on the x-axis)

The best upper bound on the number of flip events is $\mathcal{O}(n^3)$, but no input is known to cause more than $\mathcal{O}(n^2)$ flip events. While we provide no new theoretical insight on the maximum number of flip events, our tests provide strong experimental evidence that we can indeed expect a linear number of flip events for all practical data. Fig. 6 shows the number of flip events per input vertex (y-axis), for different input sizes n arranged on the x-axis. On average, our algorithm had to deal with a total of $n/4$ flip events. Over the entire set of twenty-thousand inputs only a dozen cases, mostly 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

Fig. 7. Runtime and memory usage behavior of CGAL, Bone, and Surfer for inputs of different sizes (x-axis). Bone and Surfer use their IEEE 754 double precision backend.

of input. For instance, a closer inspection of the test results revealed that synthetic "random" polygons generated by RPG [3] require significantly more flips than random axis-aligned polygons.

4.2 Runtime Performance Statistics

The following tests were conducted on an Intel Core i7-980X CPU clocked at 3.33 GHz, with Ubuntu 10.04. 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 [8], 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 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 [10], and against Cacciola's implementation [4] that is shipped with the CGAL library, version 4.0 [5]. Input to the latter was confined to polygonal data as the implementation cannot handle generalized PSLGs.

As can be seen in the left plot of Fig. 7, 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 an 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.

Measurements of memory use, shown in the right plot of Fig. 7, confirm the expected linear memory footprint of Surfer. Its memory consumption is similar to that of Bone, while the CGAL code exhibits quadratic memory requirements.

As stated, Surfer can use the MPFR library for extended-precision floating-point operations. Obviously, extended-precision arithmetic incurs a penalty both

in runtime and space requirements. Our tests with MPFR version 3.0.0 showed a decrease in speed by a factor of roughly $9.64 \cdot 10^{-4} p\sqrt{p} + 9$, where p denotes the MPFR precision. In particular, running Surfer with an MPFR precision of 100 takes about ten times as long as running it in IEEE 754 mode; at a precision of 1000 the slow-down factor is already 40. (Likely, the slow-down follows a $p\sqrt{p}$ law due to the increased complexity of doing multiplications with a larger number of digits.) The memory requirement increases linearly as the MPFR precision is increased. Roughly, the blow-up factor is modeled by $3.66 + 9.72 \cdot 10^{-3} p$.

5 Conclusion

We explain how the triangulation-based straight-skeleton algorithm by Aichholzer and Aurenhammer can be extended to make it handle real-world data on a finite-precision arithmetic. While the basic algorithm is simple to implement, all the subtle details discussed in this paper increase its algorithmic and implementational complexity. However, extensive tests clearly demonstrate that the resulting new code Surfer is the fastest straight-skeleton code currently available. In particular, our tests provide experimental evidence that only a linear number of flips occurs in the kinetic triangulation of practical data, allowing Surfer to run in $\mathcal{O}(n \log n)$ time in practice, despite of an $\mathcal{O}(n^3 \log n)$ theoretical worst-case complexity.

References

1. O. Aichholzer, D. Alberts, F. Aurenhammer, and B. Gärtner. Straight Skeletons of Simple Polygons. In *Proc. 4th Internat. Symp. of LIESMARS*, pages 114–124, Wuhan, P.R. China, 1995.
2. O. Aichholzer and F. Aurenhammer. Straight Skeletons for General Polygonal Figures in the Plane. In A.M. Samoilenko, editor, *Voronoi's Impact on Modern Science, Book 2*, pages 7–21. Institute of Mathematics of the National Academy of Sciences of Ukraine, Kiev, Ukraine, 1998.
3. T. Auer and M. Held. Heuristics for the Generation of Random Polygons. In *Proc. Canad. Conf. Comput. Geom. (CCCG'96)*, pages 38–44, Ottawa, Canada, August 1996. Carleton University Press.
4. F. Cacciola. 2D Straight Skeleton and Polygon Offsetting. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012.
5. CGAL. Computational Geometry Algorithms Library. http://www.cgal.org/.
6. S.-W. Cheng and A. Vigneron. Motorcycle Graphs and Straight Skeletons. *Algorithmica*, 47:159–182, February 2007.
7. D. Eppstein and J. Erickson. Raising Roofs, Crashing Cycles, and Playing Pool: Applications of a Data Structure for Finding Pairwise Interactions. *Discrete Comput. Geom.*, 22(4):569–592, 1999.
8. GNU. The GNU MPFR Library. http://www.mpfr.org/.
9. S. Hanke, T. Ottmann, and S. Schuierer. The Edge-Flipping Distance of Triangulations. *J. Universal Comput. Sci.*, 2:570–579, 1996.
10. S. Huber and M. Held. Theoretical and Practical Results on Straight Skeletons of Planar Straight-Line Graphs. In *Proc. 27th Annu. ACM Sympos. Comput. Geom.*, pages 171–178, Paris, France, June 2011.