Computational Geometry: Theory and Applications 96 (2021) 101760

Contents lists available at ScienceDirect

Computational Geometry: Theory and Applications

www.elsevier.com/locate/comgeo

Implementing straight skeletons with exact arithmetic: Challenges and experiences ☆

Günther Eder, Martin Held *, Peter Palfrader

Universität Salzburg, FB Computerwissenschaften, 5020 Salzburg, Austria

ARTICLE INFO

Article history:
Received 1 September 2020
Received in revised form 22 January 2021
Accepted 19 February 2021
Available online 25 February 2021

Dataset link:
https://github.com/cgalab/monos

Dataset link:
https://github.com/cgalab/surfer2

Keywords:
Straight skeleton
Weighted
Implementation
Experiments
CGAL

ABSTRACT

We present CGAL implementations of two algorithms for computing straight skeletons in the plane, based on exact arithmetic. One code, named SURFER2, can handle multiplicatively weighted planar straight-line graphs (PSLGs) while our second code, MONOS, is specifically targeted at monotone polygons. Both codes are available on GitHub. We discuss algorithmic as well as implementational and engineering details of both codes. Furthermore, we present the results of an extensive performance evaluation in which we compared SURFER2 and MONOS to the straight-skeleton package included in CGAL. It is not surprising that our special-purpose code MONOS outperforms CGAL's straight-skeleton implementation. But our tests provide ample evidence that also SURFER2 can be expected to be faster and to consume significantly less memory than the CGAL code. And, of course, SURFER2 is more versatile because it can handle multiplicative weights and general PSLGs as input. Thus, SURFER2 currently is the fastest and most general straight-skeleton code available.

1. Introduction

Straight skeletons were introduced to computational geometry by Aichholzer et al. [2]. Suppose that the edges of a simple polygon P move inwards with unit speed in a self-parallel manner, thus generating mitered offsets inside of P. Then the *(unweighted) straight skeleton* of P is the geometric graph whose edges are given by the traces of the vertices of the shrinking mitered offset curves of P; see Fig. 1, left. The process of simulating the shrinking offsets is called *wavefront propagation*. Straight skeletons are known to have applications in diverse fields, with the modeling of roof-like structures being one of the more prominent ones [22,17,18]. We refer to Huber [19] for a detailed discussion of typical applications.

As a first generalization of straight skeletons, the *multiplicatively weighted straight skeleton* was introduced early on by Aichholzer and Aurenhammer [1] and then by Eppstein and Erickson [15]. In the presence of multiplicative weights, wavefront edges no longer move at unit speed. Rather, they move at different speeds: Every edge of P is assigned its own constant speed; see Fig. 1. Although weighted straight skeletons have been used in applications for years, their characteristics were studied only recently by Biedl et al. [5,6].

Held and Palfrader [17] define an *additively weighted straight skeleton* as the geometric graph whose edges are the traces of vertices of the wavefronts over the propagation period, with additive weights being assigned to the edges of P. The

☆ Work supported by Austrian Science Fund (FWF): Grants ORD 53-VO and P31013-N31.
* Corresponding author.
E-mail addresses: geder@cs.sbg.ac.at (G. Eder), held@cs.sbg.ac.at (M. Held), palfrader@cs.sbg.ac.at (P. Palfrader).

https://doi.org/10.1016/j.comgeo.2021.101760

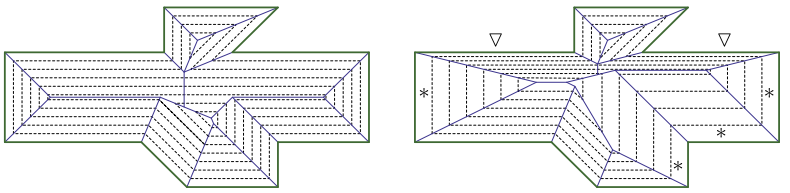


Fig. 1. Left: The (unweighted) straight skeleton (in blue) plus a family of wavefronts (dashed) for the green polygon. Right: The weighted straight skeleton for the case that edges marked with $*$ have twice the weight and edges marked with $\triangledown$ have half the weight of the unmarked edges. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

presence of additive weights means that the edges of P start to move at different points in time. In [17,18] they study straight skeletons of planar straight-line graphs (PSLGs) with a combination of both additive and multiplicative weights. (A PSLG is an embedding of a planar graph such that all edges are straight-line segments that do not intersect pairwise except at common end-points.)

The straight-skeleton algorithms with the best worst-case bounds are due to Eppstein and Erickson [15] and Vigneron et al. [10,26]. These algorithms seem difficult to implement. Indeed, progress on implementations has been rather limited so far. Felkel and Obdržálek [16] describe a simple straight-skeleton algorithm but it turned out to be flawed [27]. The first comprehensive code for computing straight skeletons was implemented by Cacciola [9] and is shipped with CGAL [25]. It is based on the algorithm by Felkel and Obdržálek [16], but was modified significantly to make it work correctly [8]. It handles polygons with holes as input.

The straight-skeleton code Bone by Huber and Held [21] handles PSLGs as input and runs in $O(n \log n)$ time and $O(n)$ space in practice. However, it is not capable of handling weighted skeletons. As a first step to extend Bone to weighted skeletons, Eder and Held [11] generalize motorcycle graphs to weighted motorcycle graphs. Still, while this is a fascinating research area of its own, this does not yet provide an avenue for an actual implementation.

2. Contribution

Palfrader et al. [24] describe a prototype implementation of a straight-skeleton code named Surfer. Since Surfer is the fastest implementation of unweighted straight skeletons known to us, it was the natural candidate for a re-implementation and extension to multiplicative weights. As the handling of degenerate cases had been fairly tricky for unweighted input, we decided to base the new implementation, Surfer2, on exact arithmetic. (However, Surfer2 has an option to run it on conventional floating-point arithmetic.) In order to be able to compare Surfer2 to a special-purpose algorithm for computing straight skeletons of monotone polygons, which is known to have an $\mathcal{O}(n \log n)$ worst-case complexity, we also implemented the algorithm by Biedl et al. [4]. This implementation was named Monos.

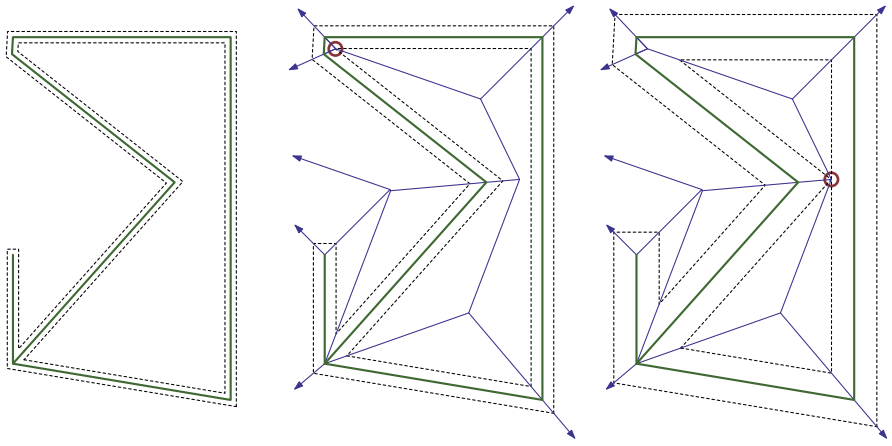
We subjected our implementations to extensive practical tests. Summarizing, Surfer2 is slower than Cgal's straight-skeleton code for small data sets, of about the same speed for polygons that contain 1000 vertices, and 100 times faster for polygons with 10 000 vertices. The memory footprint of Surfer2 is linear while Cgal's code consumes a quadratic amount of memory. For monotone polygons, our code Monos achieves a speed-up of about 1.5 compared to Surfer2. Our tests showed clearly that engineering considerations do yield practical speed-ups and that there is a significant price to pay for the use of exact arithmetic.

3. Straight skeleton and wavefront propagation

While every input edge of a polygon emanates a wavefront edge only to one side, namely to the interior of the polygon, PSLG edges emanate wavefront edges on both sides. As in the case of polygons, all wavefront edges are joined-up along angular bisectors of the input edges. In order to handle *terminal* input vertices, i.e., vertices of degree one, at each such vertex v one additional wavefront edge is created that moves away from v and its corresponding PSLG edge e in a direction orthogonal to e; see the left-most vertical segment in Fig. 2a. This way, the wavefronts form closed polygons even in the presence of degree-one input vertices.

Initially, the wavefront coincides with the input edges. If sufficiently little time has elapsed after the start of the wavefront propagation such that no event has occurred yet, then the wavefront forms a mitered offset of the PSLG. The polygons of this mitered offset partition the plane into two (possibly disconnected or multiply-connected) areas: one area that has been swept by the wavefront and another area that has not yet been reached. As time increases and the propagation continues, the area swept by the wavefront gets larger and larger, and wavefront edges will shrink to zero length, thus collapsing in *edge events*; see Fig. 2b. Furthermore, vertices may move into the interior of non-incident wavefront edges in *split events*; see Fig. 2c. Every split event splits one wavefront edge. Depending on the topology of the not-yet swept region affected, such an event will also split the region into two polygons or reduce the number of holes of the region.

This wavefront-propagation process continues until no more events happen. Then the straight skeleton is the planar graph whose edges are the traces of wavefront vertices during the propagation. Note that some wavefront vertices on the



(a) initial wavefront. (b) edge event. (c) split event.

Fig. 2. Wavefronts (dashed) at specific times for a PSLG (green), straight skeleton (blue). At an edge event, an edge of the wavefront shrinks to length zero, while at a split event a reflex vertex of the wavefront crashes into an edge of the wavefront, causing the edge (and the wavefront) two be split into two parts.

outer region will continue to escape to infinity; these vertices trace out rays that form unbounded arcs of the straight skeleton.

Simulating the wavefront propagation is a common building block of several straight-skeleton algorithms. These algorithms differ primarily in how they find the next event. All future events already known are usually maintained in a priority queue of event times. It is standard to use a min-heap to store the times of those future events. Of course, one may also associate additional information with every event time, such as references to affected wavefront edges and the location where the event will happen.

4. Straight skeleton of a monotone polygon

4.1. Monotone algorithm

Biedl et al. [4] describe an $\mathcal{O}(n \log n)$ time algorithm to compute the straight skeleton of a simple n-vertex monotone polygon $\mathcal{P}$. Without loss of generality we assume $\mathcal{P}$ to be x-monotone. Their algorithm consists of two steps: (i) The polygon $\mathcal{P}$ is split into an upper and lower monotone chain, and the straight skeleton of each chain is computed individually. (ii) The final straight skeleton $\mathcal{S}(\mathcal{P})$ is obtained by merging these two straight skeletons.

We employ a classical wavefront propagation to obtain the straight skeleton of a monotone chain in $\mathcal{O}(n \log n)$ time [4]. The monotonicity of the chains guarantees that every change of the wavefront topology is witnessed by an edge event. That is, split events cannot occur.

In the second step, the skeletons of the upper and lower chains are merged to form $\mathcal{S}(\mathcal{P})$. This merge is based on a left-to-right traversal of the two chains and their respective straight-skeleton faces. Starting at the leftmost vertex of both chains, the angular bisector between the incident edges is constructed. It intersects arcs of both the top and bottom straight skeleton. We stop at the first intersection reached and modify the respective straight skeleton locally by creating a node. Then the next bisector is constructed between the two edges that induce the faces of the upper and lower skeleton we are currently in, see Fig. 3. This process is repeated until the rightmost vertex of $\mathcal{P}$ is reached, thus obtaining the final skeleton $\mathcal{S}(\mathcal{P})$.

4.2. Implementational details

Monos uses CGAL's `Exact_predicates_exact_constructions_kernel_with_sqrt` algebraic kernel, which is backed by CORE's `Core::Expr` exact number type.

Intersection computations: A major part of the merge step are intersection tests and intersection computations between bisectors and skeleton arcs: Careful engineering resulted in substantial performance improvements for these intersection tests. Initially, we applied CGAL's `do_intersect` and `intersection` rather naively to an arc segment (seen as a straight-line segment) and a bisector. However, explicitly deciding whether the end-points of an arc segment lie on different sides of the supporting line of a merge bisector is sufficient to decide whether an intersection occurs. Once we know that an intersection occurs then we apply CGAL's `intersection` routine to the supporting lines of the arc and the bisector. A test

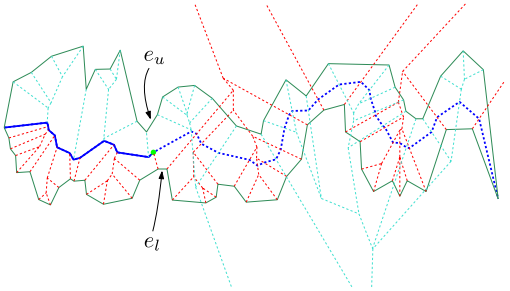


Fig. 3. The straight skeletons of the lower chain (dashed red) and of the upper chain (dashed turquoise). The merge (in blue) has proceeded to the point shown in green. The next bisector is defined by the edge e_l of the lower chain and the edge e_u of the upper chain.

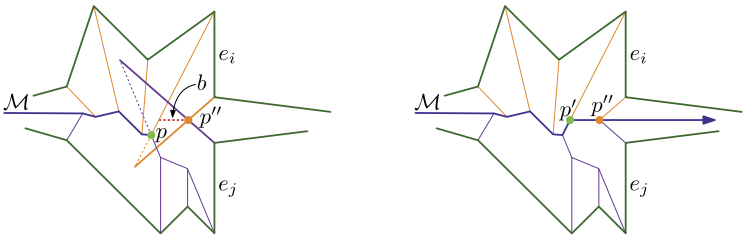


Fig. 4. Left: Constructing the merge chain $\mathcal{M}$ at a specific point p where collinearities have to be handled. Right: Looking ahead lets us identify p' as start of the next horizontal bisector.

on more than one thousand input polygons with at least 10 000 vertices each shows average runtime savings of about 9% when using the latter method.

Collinear edges: Input that contains collinear edges needs special care. The crux is that no "direction" need be implied by a wavefront vertex that traces out a bisector. Assume that two collinear wavefront edges become adjacent during the computation of the straight skeleton of a monotone chain. At the time of the event we have a straight-skeleton node p at which the event occurs. Hence we define the bisector of the two collinear edges to be perpendicular to them and to start at p, thereby moving into the same direction as the wavefront edges.

During the merge step, it is less obvious how to proceed in such a case. In Fig. 4, left, we see that the left-to-right merge has arrived at the point p. The next bisector, b, is defined by the input edges e_i and e_j. If the merge had progressed from right to left, then b would be horizontal and it would start at the point p'', in analogy to how this situation would be handled if it occurred during the construction of the straight skeleton of a monotone chain. See the horizontal red dotted segment, b, in Fig. 4, left: This bisector need not pass through p but can arrive anywhere along the boundary of the face of e_i or the face of e_j. Hence, we need to look ahead and find the point p' where b intersects one of the two boundaries. Note that this "look ahead" does not break the algorithm's complexity as we only have to walk along the boundaries of the two faces we stepped into. At some point, the boundaries intersect, which is the end-point of b. If b is not incident to p, then we need to add the start node p' of b as well; see Fig. 4, right. Afterwards we can proceed with the standard merge.

Min-heap: Initially, Monos employed C++'s `std::set` as a priority queue due to the requirement to update elements. This structure tends to be implemented as a red-black tree and provides all operations required at appropriate asymptotic costs. However, profiling Monos showed that a significant amount of time was spent in queuing operations. Therefore, we switched to a self-developed binary min-heap already in use in Surfer2. Performance tests showed an average performance gain of 5%.

5. Weighted straight skeleton of a PSLG

5.1. Triangulation-based algorithm

Aichholzer and Aurenhammer [1] describe an algorithm for computing the straight skeleton of general PSLGs in the plane. It carries over to multiplicatively weighted input in a natural way, provided that all weights are positive. (Biedl et al. [5] show that most of the theory falls apart if negative weights are allowed.) Their approach constructs the straight skeleton by simulating the wavefront propagation. As the wavefront sweeps the plane, they maintain a kinetic triangulation of that part of the plane which has not yet been swept. This triangulation is obtained by triangulating the area inside the convex hull of all wavefronts. Furthermore, all edges of this convex hull are linked with a dummy vertex at infinity.

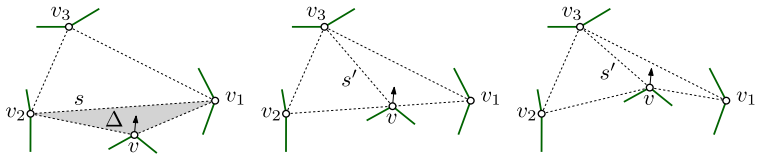


Fig. 5. Kinetic triangulation before, after processing, and some time after a flip event where reflex wavefront vertex v moves over the triangulation edge s as triangle Δ collapses. In order to keep the kinetic triangulation a valid triangulation, we need to flip s to s', the other diagonal in the quadrilateral formed by v, v_1, v_3, v_2.

The area of each triangle of this kinetic triangulation changes over time as its vertices, which are vertices of the wavefront, move along angular (straight-line) bisectors of the input edges. Since each vertex moves at constant velocity, the area of each triangle is a quadratic function in time. As long as no triangle collapses to zero area these triangles serve as certificates for the validity of the kinetic triangulation: The wavefront does not change combinatorially while no triangle collapses. Every change in the topology of the wavefront, i.e., every edge event and every split event, is witnessed by a triangle collapse. Of course, the roots of the quadratic functions give the collapse times of the triangles. These collapse times serve as keys in a priority queue of events that need to be handled.

Note that not all triangle collapses witness a corresponding event of the wavefront. Some triangle collapses correspond to internal events only, where the triangulation changes as a wavefront vertex moves over a triangulation diagonal. Aichholzer and Aurenhammer [1] call this type of event a *flip event* because it merely requires the flip of a triangulation diagonal; see Fig. 5.

The number of edge and split events is linear because the straight skeleton has a linear combinatorial complexity. However, no better bound than $O(n^3)$ is known for the maximum number of flip events for an n-vertex PSLG, with a theoretical worst-case lower bound of $\Omega(n^2)$ [19]. Consequently, the algorithm's worst-case runtime is bounded by $\Omega(n^2 \log n)$ and by $\mathcal{O}(n^3 \log n)$. Our previous work [24] showed that this algorithm runs in $\mathcal{O}(n \log n)$ time in practice when using IEEE 754 arithmetic. In the sequel we present our implementation, SURFER2, of this algorithm based on exact arithmetic, and with support for weighted input.

5.2. Implementational details

SURFER2 uses some of CGAL's geometric primitives, such as Points and Segments. It defaults to using CGAL's `Exact_predicates_exact_constructions_kernel_with_sqrt` algebraic kernel, which is backed by CORE's `Core::Expr` exact number type. SURFER2 uses little of CGAL's advanced data structures. One exception is the use of CGAL's `DCEL` arrangement [23], which was chosen for the purpose of presenting the straight skeleton to the user in a data structure they may be more familiar with. Further, we apply CGAL's constrained Delaunay triangulation package [28] for finding the initial triangulation; see below.

In addition to a library API, SURFER2, like MONOS, provides both a command line interface and a GUI [14]. GraphML [7] is our input and output format, as it allows weighted input. We also provide scripts to convert between GraphML and other formats, such as Wavefront `.obj` or `.ipe` files.

Data structures: To represent the wavefront and the area not yet swept by the wavefront, our implementation explicitly has objects of kinetic triangles, wavefront edges, and wavefront vertices. For each kinetic triangle, we store pointers to the three incident wavefront vertices, and on each side we either have a pointer to the neighboring triangle or a pointer to the incident wavefront edge. Every wavefront edge has references to its two incident wavefront vertices as well as a pointer to the incident kinetic triangle. Wavefront vertices trace out straight skeleton arcs during the propagation period, moving along the bisector of input edges. Every such wavefront vertex stores pointers to its two incident wavefront edges, i.e., the edges that emanate from the input edges defining the bisector along which the vertex moves. Furthermore, it stores the time when it came into existence, and, if it has already been stopped by an event, the time when it was stopped. Additionally, each vertex caches its velocity (as a function of its incident edges), its start location, and also its stop location (if applicable).

We start out by constructing a constrained Delaunay triangulation of the input graph using CGAL's 2D-Triangulation package [28]. From this initially static triangulation we construct the kinetic triangulation just outlined: Neighborhood relations between triangles from the initial triangulation are copied to the kinetic triangulation for triangles that are not split by a constraint. At each constraint, we initialize two wavefront edges, one moving in each direction, that are incident to one kinetic triangle each. At terminal vertices of an input graph, we need to set up the kinetic triangulation such that it can support the wavefront edge e that is emanated from this terminal vertex. (Recall that terminal vertices send out a wavefront edge that moves away from the vertex, orthogonally to the vertex's incident PSLG edge.) Therefore, we scan around the vertex to find an appropriate neighbor along a triangulation edge in the halfspace that e moves into. We create a degenerate, zero-area triangle in the place of this triangulation edge, updating the neighborhood information in the incident triangles accordingly. This new triangle has the initially zero-length wavefront edge e as one of its edges. At first, e coincides with the location of the terminal vertex but then, when the wavefront propagation commences, moves away from the vertex

orthogonally at unit speed and grows in length, causing the triangle to have positive area. Lastly, we initialize the wavefront vertices and compute their velocities based on their incident wavefront edges.

The three types of explicit objects, kinetic triangles, wavefront edges, and wavefront vertices, have different "persistence" properties. Triangle objects are only created in the setup phase just described; no new triangles are ever instantiated after the initial kinetic triangulation has been set up. A triangle's neighbors, incident edges, and incident vertices may change during the propagation when events are processed, and this usually also means that the collapse time of the triangle needs to be recomputed. As such, our priority-queue implementation has to support key-change operations. When a triangle collapses in an edge or split event, it will actually vanish for good, unlike in flip events. Therefore, we tag it as *dead*, set its collapse-time key to infinity in the priority-queue, and update incidences of the neighbors as required. The triangle object still exists, but it is of no relevance during the rest of the algorithm's run.

Wavefront edges are the edges of the wavefront that at any time form the polygonal boundary between the area swept and the area not yet swept by the wavefront. In the setup phase, we create two new edges per input edge, with one new edge moving to the left and one new edge moving to the right of the input edge. In addition, one edge is created for every terminal vertex of the input. Edges are never deleted, however, just like triangles, edges can be marked *dead*. This happens either when they collapse to zero length and, thus, vanish from the polygonal boundary, or also when they get partitioned in a split event in which case the original edge gets replaced by two or more new wavefront edge objects. The incident vertices of each wavefront edge may change over time.

Kinetic vertices, the vertices of the wavefront, have a fixed velocity (i.e. directed speed), starting time, and starting position. During event handling, we may *stop* a vertex at its current position and at the current time, and then create a new kinetic vertex object. As such, each kinetic vertex represents some arc of the straight skeleton as its being traced out. If the vertex has stopped already, it means the arc of the straight skeleton is an edge. If, even after all events have been handled, a vertex is not marked stopped, then that represents an arc that is a ray escaping to infinity. Each vertex also has pointers to the next and previous vertex for both of its incident straight skeleton faces, thereby forming a (non-standard) doubly-connected edge list. This doubly-connected edge list has full edges instead of half-edges and these full edges are directed by how they were traced out, not cyclically around faces as would be common for half-edges. After all events have been handled, we build a more standard variant of a doubly-connected edge list to provide it to the user of the Surfer2 library.

Event types and event queue: The event queue is a binary min-heap whose elements are kinetic triangles, with an order imposed by the next event each triangle witnesses. An unbounded triangle, which is defined by the vertex at infinity and one edge of the convex hull, serves not only as a witness for its edge collapsing but also for one of its vertices leaving the convex hull. For each bounded triangle, we store the time of its next collapse.

The heap is realized as an array that is created during the setup-phase, which is feasible since no new triangles are created after the initial setup and so the heap will never need to accommodate more entries. Each triangle stores an index indicating where in the heap it is currently located, and update operations in the heap (bubble up/down) take care to also update this information in each affected triangle. This enables cheap key changes when a triangle's collapse time gets updated: We find the triangle in the heap in constant time, and then restore the heap property by bubbling the element up or down at most logarithmic cost.

Finding the collapse time of a triangle is straightforward in theory. (Recall that it is a root of a second-degree polynomial given by the signed determinant of its moving vertices.) Checking when a vertex leaves the convex hull can be achieved by considering triangles of three subsequent convex-hull vertices; a collapse of such a triangle indicates a change. In practice we would like to know not only the time but also the type of event. This helps to ensure we make progress with the wavefront propagation. Progress is guaranteed if we process an event that is a split event or an edge event: There is only a linear number of such "real" events because each event effectively causes a collapsed triangle to be removed from the triangulation. The crux comes to light when investigating flip events. A flip event happens when a wavefront vertex becomes incident on a triangulation diagonal. We consider the quadrilateral formed by the two triangles along this diagonal, and flip the diagonal within this quadrilateral. If the other triangle also collapsed at precisely the same time, then after this flip we still have two triangles that collapse right now, and we might do another flip that re-establishes the previous configuration; cf. Fig. 6.

To handle multiple events that occur at the same time we apply a twofold strategy: First, we always prioritize "real" events over flip events. (This can be done with no additional cost by using a secondary key for the priority queue.) Second, we need to impose an order on flip events. Consider a maximal set $\mathcal{T}$ of neighboring triangles that all collapse at the same time to line segments on the same supporting line, without any triangulation diagonal shrinking to zero length. Note that a flip in a triangle will always cause its longest edge to be flipped. Let T be the triangle that has the longest edge e among the triangles of $\mathcal{T}$. If the other triangle, T', that is incident at e does not belong to $\mathcal{T}$, then it has positive area, and performing the flip will reduce the number of elements in $\mathcal{T}$ and we have made progress. (This is the case in Fig. 6 in the upper configuration, where we always flip T_1 first as it has the longest edge. It flips to the top towards some other triangle that is not shown in the figure, thus escaping from the flip-event loop.) If, however, $T' \in \mathcal{T}$, then performing the flip will cause the longest edge within $\mathcal{T}$ to become strictly smaller: Recall that e is the longest edge of T. Let ABC be the vertices of T, and let ABD be the vertices of T', with e being the line segment $\overline{AB}$. The edge e must also be the longest edge of T' as, by assumption, e is the longest edge of all collapsed triangles of $\mathcal{T}$. Thus, both C and D lie in the interior of e, and

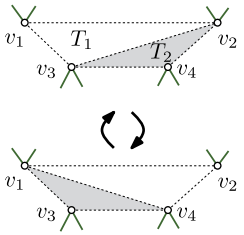


Fig. 6. Potential flip-event loop when, at time t, the vertices v_1 through v_4 all become collinear: Flipping the highlighted triangle T_2 in the top configuration will cause the bottom configuration, where a flip back to the prior setting is possible. If, however, in the top configuration we flip T_1 first, no loop happens.

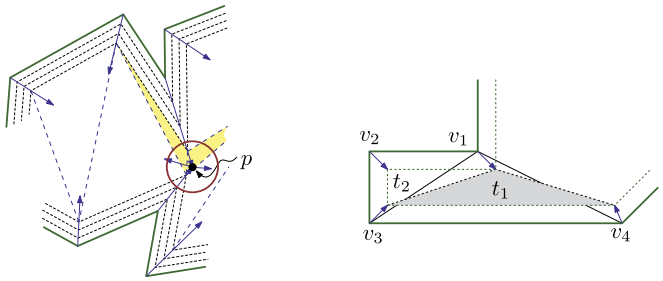


Fig. 7. Left: The two shaded triangles are witnesses for an event at p, where two reflex vertices move into each other. Right: Infinitely fast vertices get created when the triangles collapse. If t_1 is processed first, an infinitely fast vertex is created at the locus of v_1 moving to the left, and another one at the locus of v_2 and v_3 moving to the right, both incident to t_2. If t_2 is processed first, only one infinitely fast vertex is created at the locus of v_2 and v_3 moving to the right.

so replacing e, i.e., $\overline{AB}$, with $\overline{CD}$ reduces the length of the longest edge of triangles in $\mathcal{T}$.) Hence, a monotonicity argument implies that we make progress also in this case.

Another type of event that shows up in practice if the input need not be in general position is given by non-incident vertices that coincide at some point in time. Topologically, this is equivalent to a split event as previously non-incident wavefront portions become incident. From the viewpoint of a kinetic triangulation this event looks more like an edge event, with the exception that it is not a wavefront edge that collapsed but a triangulation diagonal. Handling this event is thus similar to handling two edge events, with one event per triangle incident at the collapsed diagonal. Fig. 7, left, shows such an event.

When parallel wavefront elements move into each other, events happen as triangles collapse. The wavefront edges that comprise these parallel wavefront elements are incident to different kinetic triangles. As we process these collapsed triangles, one after the other, we want to leave the kinetic triangulation in a consistent state between events. Generally, this means creating new kinetic vertices to replace old ones as edges collapse and splits happen, which involves computing the velocity of the new kinetic vertex. The velocity is such that the wavefront vertex remains on the intersection of the wavefront edges as they propagate, i.e., it points along the angular bisector of the incident edges and has the appropriate speed. However, what is the velocity of a wavefront vertex between opposing wavefront edges that overlap and span an angle of zero? The limit, as the angle approaches zero, is infinite speed with a direction along the wavefront edges. Therefore, in our implementation we have the concept of infinitely fast vertices. These get created when opposing wavefront edges become incident and share a vertex; cf. Fig. 7, right. Whenever we create an infinitely fast vertex, the triangle that is incident to this vertex is processed next as we consider it to have an event immediately. If more than one triangle is incident to an infinitely fast wavefront vertex v, then we pick the one that has the shorter constraint incident to v.

During the wavefront propagation, instances may occur where three or more wavefront vertices become collinear and remain so in the course of the propagation. If these vertices are adjacent on the convex hull, then the determinant check for whether a vertex leaves the convex hull would show a zero at all times. When encountering an always-zero determinant, depending on the specific configuration, scheduling an event for “right now” may resolve the issue, e.g., by flipping a triangulation diagonal such that the three vertices involved no longer are incident to the same kinetic triangles. If three vertices on the convex hull are collinear, then such a flip would not be advisable because removing the middle vertex from the convex hull (and keeping it on the books as a non-convex-hull vertex) would cause a bounded triangle with an always-zero determinant.

Collapse time and order of events: While the actual collapse time is one of the roots of a quadratic polynomial, solving quadratics is not always necessary. Avoiding root finding for quadratic polynomials will increase accuracy when working with limited-precision data types, and it will result in less complex expression trees when working with exact numbers as provided by CORE's `CORE::Expr`. In particular, it will avoid the computation of another square root. Hence, we try to employ geometric knowledge derived from local combinatoric properties as much as this is possible. For instance, a triangle

with two incident wavefront edges will never see a flip event: If it sees an event at all, it will be the collapse of either one or both of its wavefront edges. (In the latter case the entire triangle collapses.)

Triangles with exactly one incident wavefront edge can encounter all three types of events — edge, split, and flip events — but we can determine each such event without computing the roots of a determinant: The times of split and flip events can be found by considering the distance between the vertex opposite the wavefront edge to the supporting line of the wavefront edge. This distance is linear in time and when it passes through zero, we either have a flip or split event as the vertex comes to lie on the supporting line of the wavefront edge. Likewise, an unbounded triangle witnessing a wavefront edge collapse needs to consider only the linear function that models the wavefront edge's length. Unbounded triangles will not need to witness collapses of a triangulation diagonal because this event will be witnessed by the neighboring (bounded) triangle. Finding the time when a vertex leaves the convex hull may require determining the roots of a quadratic polynomial if no incident wavefront edge is on the convex hull. Otherwise, we employ the distance-to-supporting-line method again. The only bounded triangles for which we need to consider quadratic polynomials are triangles not incident to a wavefront edge: These triangles can cause flip events.

We process the events in order of their event times, breaking ties based on the event type: Events with infinitely fast vertices come first, then the other real events, and finally we process flip events and convex-hull-update events. Ties among flip events are broken by handling the event with the longest edge first. Since our implementation uses exact arithmetic provided by CORE, all these comparisons can be done exactly. Event handling generally is straightforward and consists of updating the kinetic triangulation to remain valid after the event, for real events dropping collapsed triangles, and stopping kinetic vertices and creating new ones. (Recall that we know the type of event.)

Once no event is left in the queue with a finite event time, the list of kinetic vertices with their start and stop times and positions yield the arcs of the final straight skeleton. As post processing, we construct a CGAL DCEL (doubly-connected edge list) structure from the kinetic vertices. Each kinetic vertex and each input segment corresponds to two DCEL half-edges, and each DCEL face is incident to one input segment or terminal input vertex.

Pre-emptive flips: Flip events are internal events of the kinetic triangulation, and as such they are not directly needed for the wavefront propagation process. They are always the result of a reflex vertex moving over the opposite diagonal in a triangle. As already outlined, flip events can lead to loops and thus require special handling. We tested whether we can reduce the number of flip events by restructuring triangles as soon as they are created, either initially or as the result of another event, rather than wait for a triangle to collapse before we do a flip. The very simple approach that we chose was to immediately flip a triangulation diagonal that is incident to two convex vertices if the flip would make it incident to a reflex vertex, in the assumption that this reflex vertex is then less likely to cause a flip event later.

We benchmarked this approach and found that it results in fewer flip events for about 98% of our test runs. In the worst example we got 4% more flip events, and in the best case flip events were reduced by slightly more than 10%, with a median change of 3% fewer flips. The benefit is less clear-cut when looking at runtime improvements, though. We noticed an improvement of about 1% in the median and a surprising 50% in the best case, but also a 30% slowdown in the worst case. Still, pre-emptive flips are the default set-up for SURFER2.

Component-based priority queues: Closed input loops partition the plane into connected components. (And in practice many PSLGs contain such loops.) The basic algorithm deals with events, witnessed by triangle collapses, without any consideration of the component the triangle is in. But the straight skeleton within one component is entirely independent of the straight skeleton within any other component: The different wavefronts never interact during their respective propagation processes. Hence, it would suffice to ensure that events are ordered correctly within each component but we could handle events from different components independently. Of course, in the perfect world of the Real RAM model it does not matter whether k items are to be stored in one or a few comparison-based priority queues, since handling k items takes $\Theta(k \log k)$ time one way or the other.

Once we talk about an implementation the Real RAM model is no longer applicable, though. Our implementation uses CORE's `CORE::Expr` number type as shipped with CGAL. This gives us the luxury of actually having exact arithmetic, thus making it easy to know which events happen simultaneously. The significant price to be paid is that comparisons are no longer unit-cost: Each CORE expression variable carries with it its expression tree as the entire history of how it was derived, as well as a numerical approximation with error bounds. Whenever we need to compare two numbers, CORE can calculate more digits should this be needed to ascertain the exact relationship. This takes extra time and memory. In general, such comparisons are near-instantaneous when the values are (sufficiently) different. However, they can take a very long time if the numbers are actually equal. (Profiling showed that one such comparison may even take many seconds.) For further details on this issue we refer to our discussion of experimental results in Section 6.

To avoid comparing event times from different components we assign an integer identification number to each component and use this number as a secondary key for the event ordering in the priority queue. We added instrumentation to our implementation to count how often we actually compare equal event times during the wavefront propagation process. For random input in general position, the likelihood of events in different parts of the plane to happen at the same time is zero. Our tests confirmed that for random input there is nothing to gain by per-component event ordering. However, avoiding comparing event times from different components may yield substantial savings for some input classes: A reduction of the number of simultaneous events by a quarter may result in a reduction of the runtime by a factor of five in extreme cases!

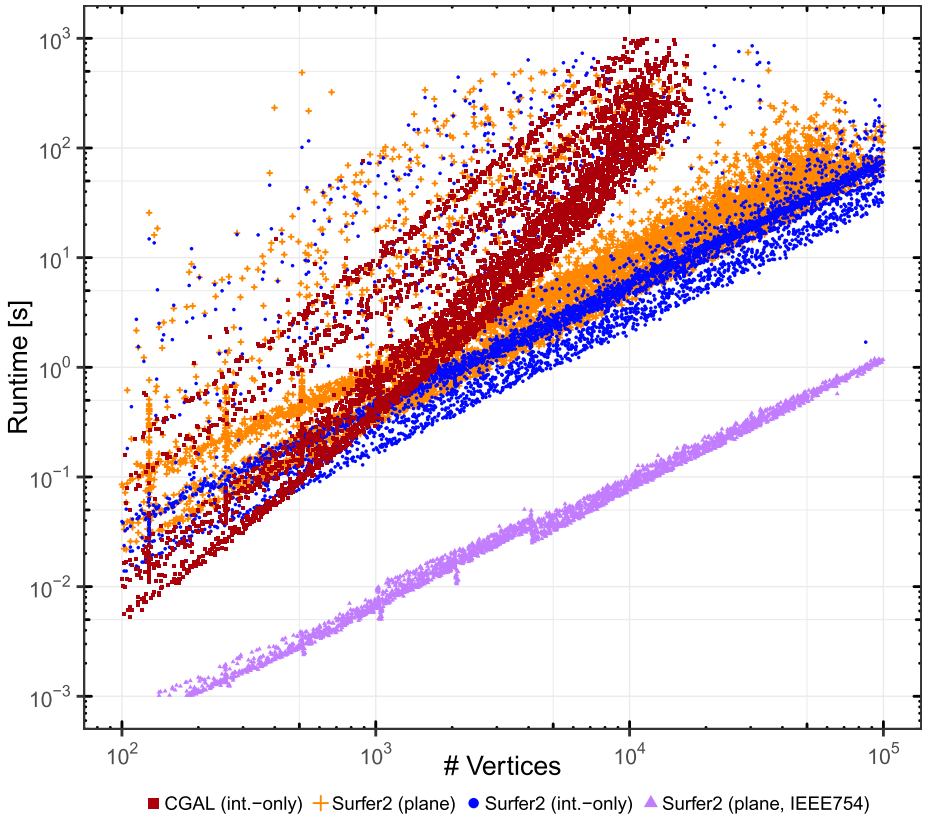


Fig. 8. Runtimes of CGAL's code and different variants of SURFER2.

Savings can be expected to occur for inputs with vertices on integer grids, or, e.g., if two components both contain tight polygonal approximations of circular arcs with the same radius.

6. Results

Setup: All runtime tests were carried out on a 2015 Intel Core i7-6700 CPU. For most of our tests, memory consumption was limited to 10 GiB. The codes were compiled with `clang++`, version 7.0.1, against CGAL 5.0 except where stated otherwise.

The default setting for CGAL's straight-skeleton package [9] is to use the exact predicates but inexact constructions kernel that ships with CGAL. The use of this kernel ensures that the straight skeleton computed is combinatorially correct, even if the locations of the nodes need not be correct. CGAL's straight skeleton package can also be run with the exact predicates and exact constructions kernel. However this causes the runtimes to increase by a factor of roughly 100. Our codes, MONOS and SURFER2, use the CGAL exact predicates and exact constructions with square-root kernel by default, as we construct wavefront vertices with velocities, and these computations involve square roots.

We tested both CGAL and SURFER2 on many different classes of polygons. Our test data consists of real-world (multiply-connected) polygons as well as of synthetic data generated by RPG [3] and similar tools. The tools and all synthetic data are available as part of the Salzburg Database project [12,13]. Overall we ran our codes on close to 30 000 data sets. For a polygon that has holes we used CGAL's straight-skeleton code that supports holes. Otherwise we used the implementation which only supports simple polygons.

Additionally, we tested both of our codes, MONOS and SURFER2, with large monotone polygons, for up to 10^6 vertices. (We did not run CGAL on these inputs due to memory constraints.) Given the lack of a sufficiently large number of monotone real-world inputs, we used RPG [3] and other tools to automatically generate thousands of monotone input polygons. We also ran SURFER2 on hundreds of real-world PSLGs. Most of our data came from GIS sources and represents road and river networks, contour lines and the like. The runtimes on those inputs are quite comparable to the runtimes for polygons. That is, the test results presented here are also representative for SURFER2's performance on real-world data.

Runtimes and memory consumption: While CGAL's code computes the straight skeleton either in the interior or the exterior of a polygon, SURFER2 can do both in one run because it treats a polygon as a PSLG. But SURFER2 can also be restricted to just the interior or the exterior of a polygon. Hence, for the plot in Fig. 8 we ran SURFER2 twice, once applied

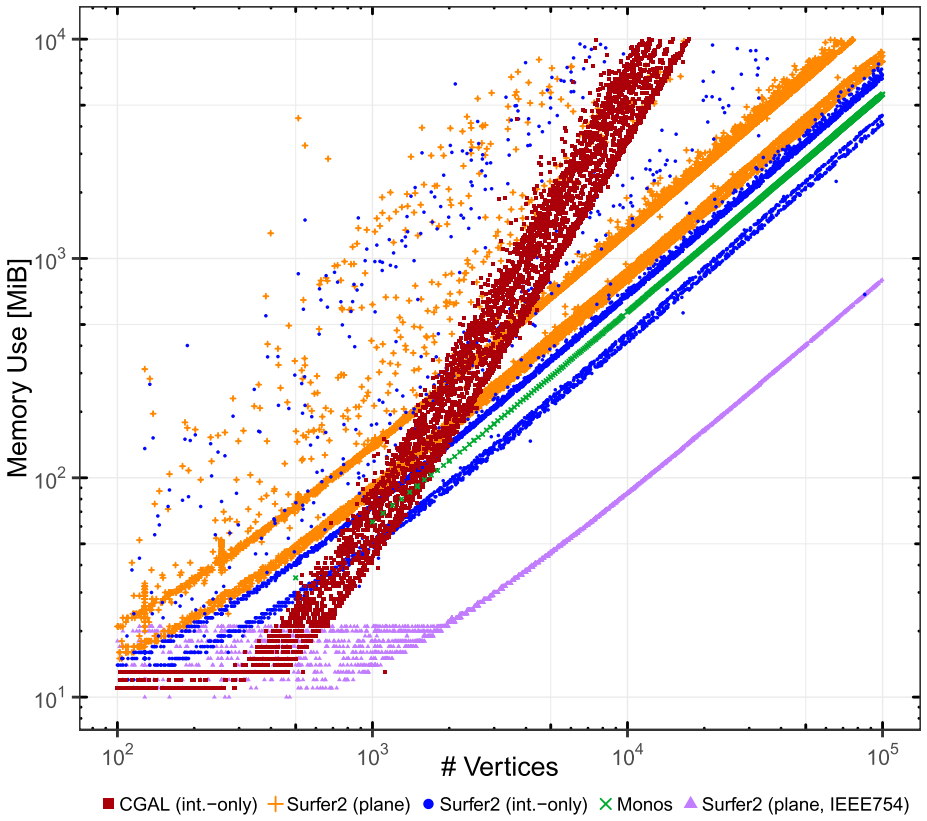


Fig. 9. Memory use of CGAL, SURFER2 variants, and MONOS.

to the entire plane and once for just the interiors of the polygons that were also handled by CGAL. Our tests make it evident that SURFER2 is significantly faster than CGAL for a large fraction of the inputs. In particular, its runtime seems to exhibit an $n \log n$ growth, compared to the clearly quadratic increase of the runtime of CGAL's code. Fig. 9 shows that SURFER2's memory consumption grows (mostly) linearly while CGAL's code requires a clearly quadratic amount of memory.

The results for CGAL's straight skeleton package were to be expected because it computes potential split events for each pair of reflex vertex and wavefront edge [19, Section 2.5.4]. Indeed, tests carried out in 2010 indicated that it requires $\mathcal{O}(n^2 \log n)$ time and $\Theta(n^2)$ space for n-vertex polygons, as discussed by Huber and Held [21]. Our test results and the CGAL release notes suggest that the same algorithm and implementation are also applied in the current CGAL 5.0, which is the version used in our tests.

As the theoretical upper bound on the runtime complexity of SURFER2 is given by $O(n^3 \log n)$ and its worst-case lower bound by $\Omega(n^2 \log n)$, the very decent near-linear practical performance is noteworthy. The worst-case lower bound comes from two independent sources: First, the fact that there is a linear number of edge and split events, and each of those may, in the worst case, affect a linear number of triangles. Indeed, Huber [19] gives a convex polygon that, with a specific triangulation, will exhibit this behavior; see Fig. 10, left. Second, the number of flip events may be quadratic in the input size too. Huber and Held [20] describe a polygon that forces a triangulation which sees a quadratic number of flip events; see Fig. 10, right.

We investigated how common such extreme cases are and counted how many triangles are affected by a single edge or split event on average and as a maximum over all our test inputs. The results can be seen in the plots in Fig. 11. As can be seen from the image on the left, the average number of affected triangles across a vast majority of inputs is bounded by a small constant. The few instances with large average numbers appear to be (rounded) straight-line approximations of circular arcs that appear in the outline of fonts or similar input. However, among all our inputs of up to 10^5 vertices, we only had one instance where the maximum number of affected triangles was greater than 10^3. (And even this number was just slightly greater than 10^3.) Thus, the cost of a real event, on average and in practice, appears to boil down to constant time operations like computing new collapse times for a small constant number of triangles plus their respective priority queue updates at logarithmic cost.

For SURFER2 to exhibit the near-linear runtime we observed, the number of flip events also needs to be limited to something significantly less than quadratic. And, indeed, our experiments show that the number of flip events across all our inputs is linear in practice: Usually we get well under one flip per input vertex; see Fig. 12.

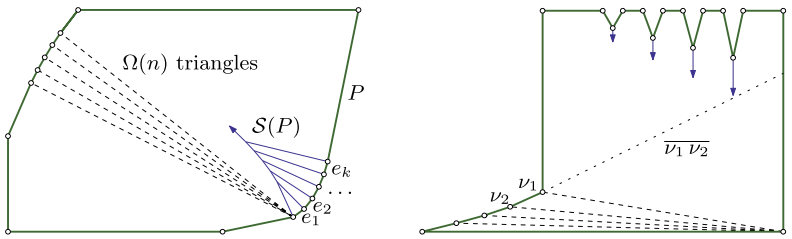


Fig. 10. (Left) The convex polygon P with the given triangulation will cause a linear number of edge events, each affecting a linear number of triangles [21]. Therefore, the triangulation-based algorithm will need at least $\Omega(n^2 \log n)$ time to construct the straight skeleton. (Right) This particular input causes a total number of $\Omega(n^2)$ flip events. By construction, no triangulation is possible that results in fewer flips [20].

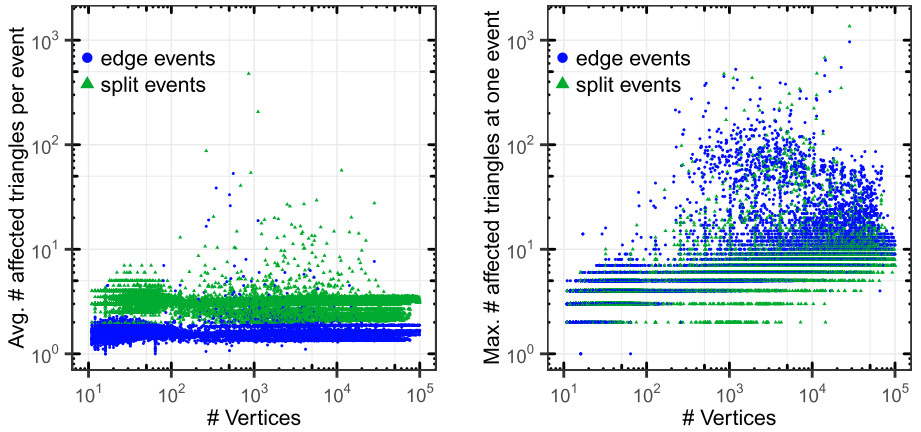


Fig. 11. Average and maximum number of triangles affected by one edge event or split event for input data of different sizes.

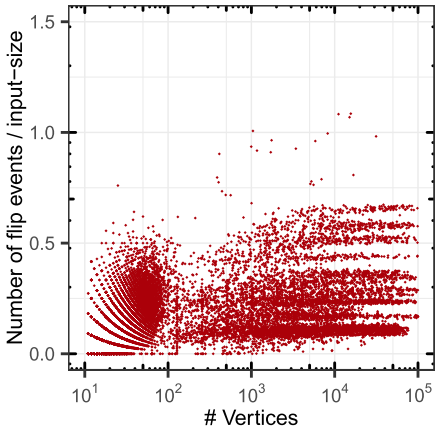


Fig. 12. Number of flip events normalized to input size.

We also ran SURFER2 with IEEE 754 double as number type instead of CORE's `CORE::Expr`. We admit, though, that SURFER2 is not (yet) fully prepared to deal with finite-precision arithmetic as it assumes that the order of the events is given reliably. (With sufficient engineering effort, however, such requirements can be relaxed and the code could be made to run reliably even with finite-precision arithmetic.) Hence, the current version of SURFER2 with the IEEE 754 backend works well on input in general position, but fails for some inputs when several events happen at exactly the same time and position. Nevertheless, we wanted to see the runtime and memory characteristics in order to get a better impression of the practical costs of using `CORE::Expr`: Therefore the plots in Figs. 8 and 9 also show results for SURFER2 with IEEE 754 arithmetic.

The $\mathcal{O}(n \log n)$ bound for the complexity of MONOS is apparent in Fig. 13, left. Given that MONOS is a special-purpose code designed specifically for handling monotone polygons, it had to be expected that it outperforms SURFER2 consistently for such input.

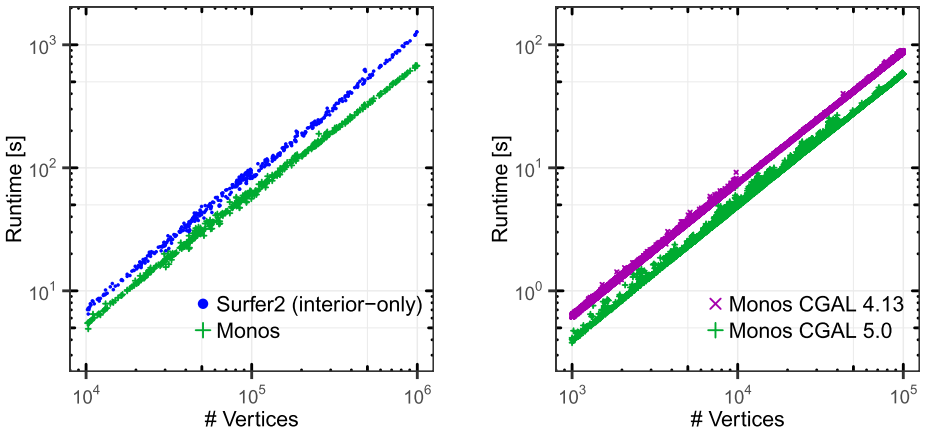


Fig. 13. Monos vs. Surfer2 on monotone inputs, and Monos with Cgal 4 vs. Cgal 5.

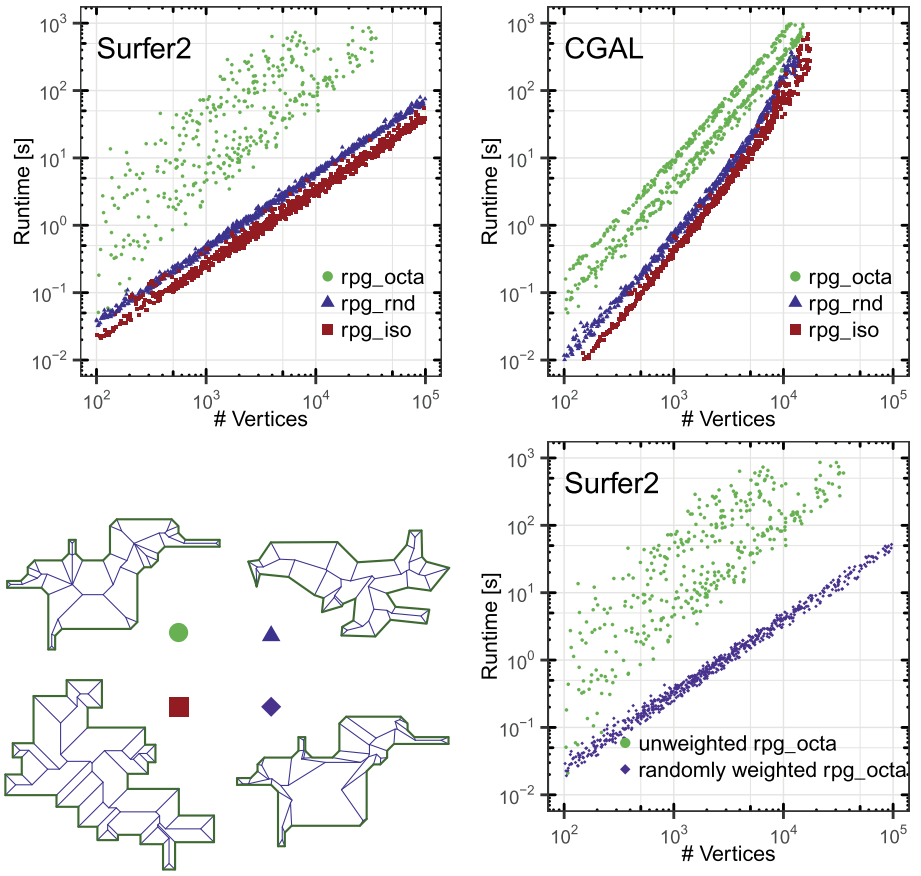


Fig. 14. Top: The effect of different input classes on the runtime of Surfer2 and on Cgal. Bottom-left: Samples for different input classes (left to right, top to bottom): Octagonal input (on integer grid), random polygon, orthogonal polygon not on integer grid, and weighted octagonal input. Bottom-right: Surfer2 runtimes for unweighted and randomly weighted octagonal input.

Dependence on input characteristics: There are outliers both in the runtime as well as in the memory consumption of Surfer2 which are clearly visible in Figs. 8 and 9. (To a lesser extent this noise is visible for Cgal, too.) Given the fact that such outliers do not show up for the IEEE 754-based version of Surfer2, there is reason to assume that this behavior is not intrinsic to Surfer2's algorithm but that it has its roots in the use of exact arithmetic. To probe this issue further we investigated which input classes trigger these outliers. Fig. 14 shows the runtimes of both codes for three different inputs classes.

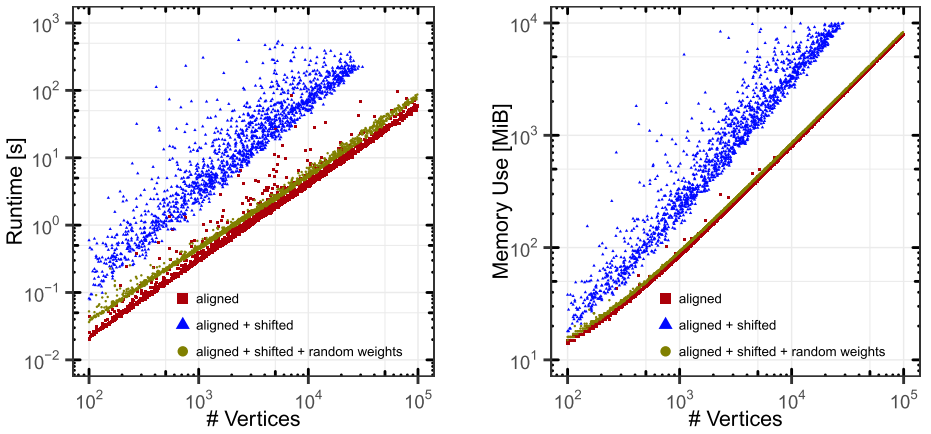


Fig. 15. Runtime and memory use of SURFER2 on orthogonal inputs with different properties. One set has its coordinates on the integer grid, the other set has all its vertices shifted by a fixed, non-integer constant. The third set contains the shifted input but assigns random weights to the input edges and computes the weighted straight skeleton. (Computations were limited to 10 GiB of memory.)

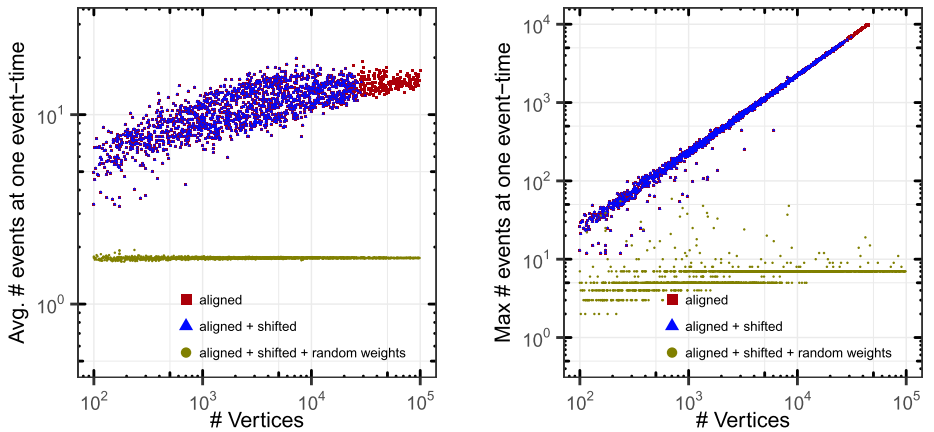


Fig. 16. Number of average (Left) and maximum number (Right) of events observed by SURFER2 at each distinct event time for the input sets of Fig. 15. Since the *aligned* and the *aligned + shifted* polygons represent the same shapes, it is not surprising that they see the same numbers of events at the same times.

The class of input that was most time-consuming to handle for all implementations were our random octagonal polygons. All polygons out of this group have interior angles that are multiples of 45°. We were surprised to see that axis-parallel input is not troublesome per se. The key difference between both groups of polygons is given by the fact that all octagonal polygons have their vertices on an integer grid while the vertices of the orthogonal polygons are (random) real numbers. Hence, for the octagonal polygons many events tend to happen at exactly the same time when opposite edges become incident in different parts of the polygon. It is apparent that the proper time-wise ordering of these events incurs a significant cost if `CORE::Expr` is used. For comparison purposes we ran the code on random simple polygons as a third class of input, with random vertex coordinates. Having any kind of co-temporal event is unlikely for those inputs.

The excessive amounts of time consumed by ordering simultaneous events became even more apparent when we studied the impact of multiplicative weights on SURFER2's runtime. As expected, a difference in the timings for weighted and unweighted random polygons was hardly noticeable. For our randomly weighted octagonal polygons we expected reduced runtimes and fewer outliers even when using exact arithmetic, due to very few truly simultaneous events. And, indeed, this was confirmed impressively by our tests; see Fig. 14.

However, note that the presence of parallel edges does not automatically increase the runtime of SURFER2, though, as further tests have revealed. In an attempt to dig further into the characteristics of event handling with exact arithmetic, we created a few thousand more orthogonal polygons, with all vertices aligned on a dense integer grid. SURFER2 performed surprisingly well, consuming approximately the same amount of time to process these data sets as other inputs of the same input size, with only a small number of outliers. However, as soon as we shifted every polygon by the constant 0.123 456 789 upwards and rightwards, both runtime and memory use went up again (Fig. 15). This is remarkable since the polygonal geometries are the same and, thus, the same numbers of co-temporal events occurred; see Fig. 16. Again, the

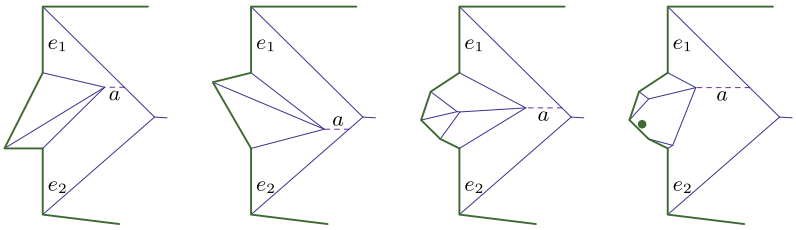


Fig. 17. The (supporting line of) straight skeleton arc a (dashed, purple) depends not only on e_1, e_2 but also on the weights (e.g., •), positions, and orientations of other, non-incident input edges.

behavior normalized as soon as random weights were introduced. The effect was also present, though to a lesser degree, when the additive value had fewer digits. It seems clear that CORE is highly sensitive to seemingly trivial changes in its input, such as apparent precision.

Experiences with CGAL: While running our tests, we also compared CGAL versions 4.13 and 5.0, as the latter was released only recently. We witnessed an improvement in the performance of our codes for the newer version; see Fig. 13, right.

We also spent a considerable amount of time on debugging SURFER2, just to find out in the end that some of our inputs trigger a bug in the `CORE::Expr` number type. This bug occurs for both CGAL 4.13 and CGAL 5.0. In specific reproducible cases the floating-point filter fails. As a result, some predicates are evaluated incorrectly, such as CGAL/CORE claiming that $0.49770 < 0.01047$. We have reported this bug to the CGAL project (Bug#4296) and the issue has since been resolved.

7. Discussion and conclusion

Our implementations MONOS and SURFER2 show that an $\mathcal{O}(n \log n)$ runtime can be achieved for the computation of straight skeletons of n-vertex monotone polygons and PSLGs, even if the edges of the PSLG are weighted by positive multiplicative weights. Engineering considerations do not improve the asymptotic behavior but help to improve the practical efficiency. Hence, our implementational efforts can be regarded as successful.

But our tests also make it evident that the use of the `CORE::Expr` number type forces one to abandon the concept of unit-cost comparisons: Multiple events that occur simultaneously have a significant impact on the practical runtime of SURFER2 if the `CORE::Expr` number type is used, while no impact is visible for the standard IEEE 754 arithmetic. It is obvious that it would pay off to detect groups of simultaneous events in a way that does not involve comparing the event times, as it is done by our current implementation. As discussed, a per-component ordering of the events in different priority queues reduces the burden that simultaneous events put on the efficiency. In our current version, we apply per-component computations only for different components induced by the input PSLG. If we could cheaply detect the splitting of a wavefront into two components and afterwards use new component numbers in future updates of the priority queue, then we might be able to exploit this idea even further. How to carry out a dynamic maintenance of the component information with little computational overhead is on our agenda for future R&D work.

A second avenue for practical improvements is given by a fine tuning of the priority queue used for storing future events. It does not come as a surprise that not all real events computed and registered in the priority queue do indeed correspond to topological changes of the actual wavefront at the times when they are predicted to occur. As a consequence, we spend time on ordering future events even if only a fraction of them will actually happen. Hence, our tests suggest that it might be better to use some form of lazy priority queue. (Standard lazy binomial heaps or Fibonacci heaps still would result in $\mathcal{O}(\log n)$ many comparisons for a delete-min operation.) After all, we do not really require a proper min-heap that obeys the heap property on every layer. Rather, all we need to know is the next event.

An alternative to constructing collapse times and then comparing them is developing predicates for ordering triangle collapses. However, this might be more involved than it appears at first glance: A triangle consists of three kinetic wavefront vertices that move along the (weighted) bisectors of pairs of input edges, which, in general, uniquely define the position of their wavefront vertex as a function of time. However, in the case of parallel, collinear wavefront edges stemming from one or more parallel, collinear input edges, their bisector is not a uniquely defined one-dimensional line. Rather, the exact trajectory of the kinetic vertex depends on the wavefront propagation that has happened so far; see Fig. 17. Nevertheless, having and using exact predicates and resorting to explicit exact computations and comparisons of the collapse times only in degenerate cases might speed up computing the straight skeleton.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Source code

Our source codes are provided on GitHub and can be used freely under the GPL(v3) license: https://github.com/cgalab/monos and https://github.com/cgalab/surfer2.

References

[1] Oswin Aichholzer, Franz Aurenhammer, Straight skeletons for general polygonal figures in the plane, in: Voronoi's Impact on Modern Sciences II, vol. 21, Institute of Mathematics of the National Academy of Sciences of Ukraine, 1998, pp. 7–21.

[2] Oswin Aichholzer, Franz Aurenhammer, David Alberts, Bernd Gärtner, A novel type of skeleton for polygons, J. Univers. Comput. Sci. 1 (12) (1995) 752–761, https://doi.org/10.1007/978-3-642-80350-5_65.

[3] Thomas Auer, Martin Held, Heuristics for the generation of random polygons, in: Proceedings of the 8th Canadian Conference on Computational Geometry (CCCG), 1996, pp. 38–44.

[4] Therese Biedl, Martin Held, Stefan Huber, Dominik Kaaser, Peter Palfrader, A simple algorithm for computing positively weighted straight skeletons of monotone polygons, Inf. Process. Lett. 115 (2) (2015) 243–247, https://doi.org/10.1016/j.ipl.2014.09.021.

[5] Therese Biedl, Martin Held, Stefan Huber, Dominik Kaaser, Peter Palfrader, Weighted straight skeletons in the plane, Comput. Geom. Theory Appl. 48 (2) (2015) 120–133, https://doi.org/10.1016/j.comgeo.2014.08.006.

[6] Therese Biedl, Stefan Huber, Peter Palfrader, Planar matchings for weighted straight skeletons, in: Proceedings of the 25th International Symposium on Algorithms and Computation (ISAAC), 2014, pp. 117–127.

[7] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, M. Scott Marshall, GraphML progress report structural layer proposal, in: Proceedings of the 9th International Symposium on Graph Drawing, 2001, pp. 501–512.

[8] Fernando Cacciola, Private email correspondence, 2010.

[9] Fernando Cacciola, 2D straight skeleton and polygon offsetting, in: CGAL User and Reference Manual, 5.0 edition, CGAL Editorial Board, 2019, https://doc.cgal.org/5.0/Manual/packages.html#PkgStraightSkeleton2.

[10] Siu-Wing Cheng, Liam Mencel, Antoine Vigneron, A faster algorithm for computing straight skeletons, ACM Trans. Algorithms 12 (3) (2016) 44:1–44:21, https://doi.org/10.1145/2898961.

[11] Günther Eder, Martin Held, Computing positively weighted straight skeletons of simple polygons based on bisector arrangement, Inf. Process. Lett. 132 (2018) 28–32, https://doi.org/10.1016/j.ipl.2017.12.001.

[12] Günther Eder, Martin Held, Steinþór Jasonarson, Philipp Mayer, Peter Palfrader, Salzburg database of polygonal data, https://doi.org/10.5281/zenodo.3784789, 2020.

[13] Günther Eder, Martin Held, Steinþór Jasonarson, Philipp Mayer, Peter Palfrader, Salzburg database of polygonal data: polygons and their generators, Data Brief 31 (2020) 105984, https://doi.org/10.1016/j.dib.2020.105984.

[14] Günther Eder, Martin Held, Peter Palfrader, Step-by-step straight skeletons, in: 36th International Symposium on Computational Geometry (SoCG 2020), in: LIPIcs, vol. 164, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Zürich, Switzerland, 2020, pp. 76:1–76:4.

[15] David Eppstein, Jeff Erickson, Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions, Discrete Comput. Geom. 22 (4) (1999) 569–592, https://doi.org/10.1145/276884.276891.

[16] Petr Felkel, Štěpán Obdržálek, Straight skeleton implementation, in: Proceedings of the 14th Spring Conference on Computer Graphics (SCCG), 1998, pp. 210–218.

[17] Martin Held, Peter Palfrader, Straight skeletons with additive and multiplicative weights and their application to the algorithmic generation of roofs and terrains, Comput. Aided Des. 92 (1) (2017) 33–41, https://doi.org/10.1016/j.cad.2017.07.003.

[18] Martin Held, Peter Palfrader, Skeletal structures for modeling generalized chamfers and fillets in the presence of complex miters, Comput-Aided Des. Appl. 16 (4) (2019) 620–627, https://doi.org/10.14733/cadaps.2019.620-627.

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

[20] Stefan Huber, Martin Held, Straight skeletons and their relation to triangulations, in: Proceedings of the 26th European Workshop on Computational Geometry (EuroCG), 2010, pp. 189–192.

[21] Stefan Huber, Martin Held, Theoretical and practical results on straight skeletons of planar straight-line graphs, in: Proceedings of the 27th Annual Symposium on Computational Geometry (SoCG), 2011.

[22] Tom Kelly, Peter Wonka, Interactive architectural modeling with procedural extrusions, ACM Trans. Graph. 30 (2) (April 2011) 14:1–14:15, https://doi.org/10.1145/1944846.1944854.

[23] Lutz Kettner, Halfedge data structures, in: CGAL User and Reference Manual, 5.0 edition, CGAL Editorial Board, 2019, https://doc.cgal.org/5.0/Manual/packages.html#PkgHalfedgeDS.

[24] Peter Palfrader, Martin Held, Stefan Huber, On computing straight skeletons by means of kinetic triangulations, in: Proceedings of the 20th Annual European Symposium on Algorithms (ESA), 2012, pp. 766–777.

[25] The CGAL Project, CGAL User and Reference Manual, 5.0 edition, CGAL Editorial Board, 2019, https://doc.cgal.org/5.0/Manual/packages.html.

[26] Antoine Vigneron, Lie Yan, A faster algorithm for computing motorcycle graphs, Discrete Comput. Geom. 52 (3) (2014) 492–514, https://doi.org/10.1007/00454-014-9625-2.

[27] Evgeny Yakersberg, Morphing Between Geometric Shapes Using Straight-Skeleton-Based Interpolation, MSc thesis, CS Dept., Technion, Haifa, Israel, 2004.

[28] Mariette Yvinec, 2D triangulation, in: CGAL User and Reference Manual, 5.0 edition, CGAL Editorial Board, 2019, https://doc.cgal.org/5.0/Manual/packages.html#PkgTriangulation2.