Computer-Aided Design and Applications

ISSN: (Print) 1686-4360 (Online) Journal homepage: www.tandfonline.com/journals/tcad20

Computing Mitered Offset Curves Based on Straight Skeletons

Peter Palfrader & Martin Held

To cite this article: Peter Palfrader & Martin Held (2015) Computing Mitered Offset Curves Based on Straight Skeletons, Computer-Aided Design and Applications, 12:4, 414-424, DOI: 10.1080/16864360.2014.997637

To link to this article: https://doi.org/10.1080/16864360.2014.997637

www.cadanda.com
Computer-Aided Design AND APPLICATIONS

Computing Mitered Offset Curves Based on Straight Skeletons

Peter Palfrader[1] and Martin Held[2]

[1]Universität Salzburg, FB Computerwissenschaften, 5020 Salzburg, Austria, palfrader@cosy.sbg.ac.at
[2]Universität Salzburg, FB Computerwissenschaften, 5020 Salzburg, Austria, held@cosy.sbg.ac.at

ABSTRACT

We study the practical computation of mitered and beveled offset curves of planar straight-line graphs (PSLGs), i.e., of arbitrary collections of straight-line segments in the plane that do not intersect except possibly at common end points. The line segments can, but need not, form polygons. Similar to Voronoi-based offsetting, we propose to compute a straight skeleton of the input PSLG as a preprocessing step for mitered offsetting. For this purpose, we extend and adapt Aichholzer and Aurenhammer's triangulation-based straight-skeleton algorithm to make it process real-world data on a conventional finite-precision arithmetic.

We implemented this extended algorithm in C and use our implementation for extensive experiments. All tests demonstrate the practical suitability of using straight skeletons for the offsetting of complex PSLGs. Our main practical contribution is strong experimental evidence that mitered offsets of PSLGs with 100 000 segments can be computed in about ten milliseconds on a standard PC once the straight skeleton is available and that our implementation clearly is the fastest code for mitered offsetting even if the computational costs of the straight-skeleton computation are included in the timings.

Keywords: Straight skeleton, mitered offset, beveled offset.

1. INTRODUCTION

1.1. Motivation and Prior Work

Offsetting is a basic geometric operation that has applications in various fields of engineering. Consider a polygon P in the plane that is closed and has no self-intersections. The interior offset curve of P with offset distance r is the set of all points within the interior of P whose (Euclidean) distance from P is exactly r. See Figure 1 for a family of offset curves inside of a polygon. It is well known that every offset curve of a polygon consists of straight-line segments and circular arcs: every offset segment is parallel to an input segment of P, and every offset arc corresponds to a reflex vertex of P, with all arcs having the same radius r. (A vertex v is called reflex if the interior angle at v is greater than π.) Since every point in the offset curve is exactly at the same distance from the source P, such an offset curve is also called a rounded or constant-radius offset curve.

A mitered offset is obtained by dropping the offset arcs and extending the offset segments in order to make them meet, see Figure 2. For mitered offsets, an offset segment is not at a fixed distance to its source segment but instead to the line supporting the source segment.

The definition of offset curves is readily extended to exterior offsets, to offsets of polygons with holes, and to one-sided offsets of open polygonal figures. It is also common to speak simply of an "offset" rather than of an "offset curve".

Conventional algorithms for offsetting proceed in four steps: (1) An elementary offset segment is generated for every segment of the input by translating it appropriately, (2) a raw offset curve is constructed by closing the gaps between the elementary offset segments, (3) all pair-wise self-intersections in the raw offset curve are computed, and (4) all invalid loops of the raw offset are removed. Several strategies have been suggested for reducing the complexity of the pair-wise intersection checks and for identifying the invalid loops correctly even if multiple intersections coincide. For further information on conventional offsetting we refer to the recent publication by Li et al. [19] and the references cited therein.

Persson, realizing the intimate connection between Voronoi diagrams and rounded offsets, sketches the first algorithm for computing an interior offset of a

Taylor & Francis
Taylor & Francis Group

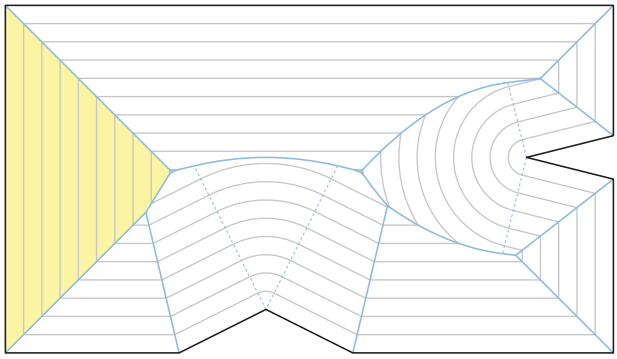

Fig. 1: Voronoi diagram (blue, solid and dotted) of an input polygon P (bold), medial axis (blue, solid arcs only), and a family of rounded offset curves (gray). One Voronoi cell is shaded yellow.

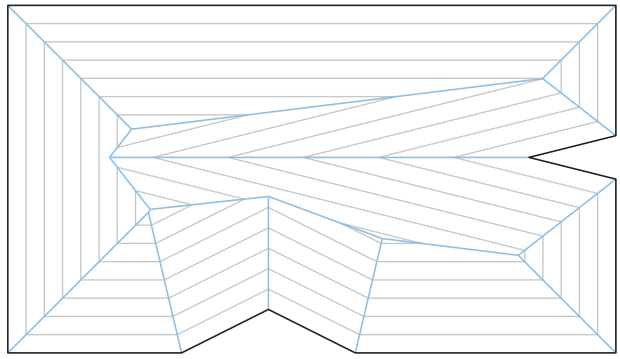

Fig. 2: The straight skeleton $\mathcal{S}(P)$ (blue) of an input polygon P (bold) is the union of traces of vertices of mitered offset curves (gray).

polygon based on its Voronoi diagram [22]. Roughly speaking, the Voronoi diagram partitions the interior of a polygon (or the entire plane) into so-called Voronoi cells around each input segment s such that any point in the Voronoi cell of s is closer to s than to any other input segment, see Figure 1. A Voronoi diagram of a polygon consists of straight-line segments and portions of conic arcs. Closely related to the Voronoi diagram in the interior of a polygon P is the medial axis, which is the set of all loci inside of P which do not have a unique closest point on P. The medial axis is a subset of the Voronoi diagram. Held [13] presents the first thorough study of computational geometry methods in Voronoi-based offsetting. We refer to recent work by Held [14] for precise definitions and a survey of algorithms for, and applications of, Voronoi diagrams of points, straight-line segments, and circular arcs.

Nowadays it is generally uncontested that Voronoi diagrams constitute the premier choice for offsetting with regard to both speed and reliability, and computing the Voronoi diagram as a preprocessing step is warranted even if only a few offsets are to be generated. Hence, it seems natural to apply a similar approach to mitered offsetting, and to use straight skeletons for mitered offsetting.

1.2. Straight Skeletons

The straight skeleton of a polygon was introduced by Aichholzer et al. [2]. Roughly, it is the geometric graph whose edges are the traces of vertices of shrinking mitered offset curves of the polygon, see Figure 2. Note that both the straight skeleton and the mitered offset curves consist of straight-line segments only.

Straight skeletons are a versatile tool in computational geometry and have found applications in diverse fields of industry and science. Tomoeda et al., for instance, use straight skeletons to create signs with an illusion of depth [26], while Sugihara uses (weighted) straight skeletons in the design of pop-up cards [24]. Aichholzer et al. [2] apply straight skeletons to roof design and terrain generation. We refer to Huber [15] for a recent survey on straight skeletons.

Contrary to Voronoi diagrams, straight skeletons are not defined relative to a metric but as the outcome of the so-called *wavefront propagation* process, which is similar in spirit to the prairie-fire analogy for Voronoi diagrams [5]: Every segment of a polygon P sends out a parallel wavefront segment that moves with unit speed to the interior of P. The wavefront segments of two adjacent polygon segments s_1, s_2 are joined by a wavefront vertex that moves along the angular bisector of s_1 and s_2. Initially, at time (i.e., offset) $t = 0$, the wavefront is identical to the input polygon P. We denote the wavefront of a polygon P at time t by $\mathcal{W}_P(t)$. At time t the (orthogonal) distance between the wavefront segments and the polygon segments equals the offset distance t. Hence, for the wavefront propagation process we feel free to mix the terms “time” and “distance”.

During the wavefront propagation, two basic types of topological changes of the wavefront occur: (1) An edge of the wavefront collapses to zero length and, thus, is removed from the wavefront. This is called an *edge-event*. (2) A vertex of the wavefront moves into the interior of a previously non-incident wavefront edge. Both the edge and the shrinking polygon will be split into two parts by this *split-event*. Even more complex interactions may occur for input that is not in general position (but has parallel edges or vertices that move toward each other), such as *multi-split-events*, when two or more vertices that are not all connected by collapsing edges become incident, or even *vertex-events*, when a new reflex vertex is born as the result of a multi-event [9,17].

At any time during the wavefront propagation, the wavefront $\mathcal{W}_P(t)$ consists of one or more mitered offset curves. The propagation process ends when all components of the wavefront have collapsed.

The *straight skeleton* $\mathcal{S}(P)$ is the geometric graph whose edges are the traces of all vertices of the wavefront over the propagation period. The edges of the

straight skeleton are called *arcs*, and its inner vertices are called *nodes*. Note that there is a one-to-one correspondence between nodes of $\mathcal{S}(P)$ and events of the wavefront propagation process, and that the leaf-vertices of $\mathcal{S}(P)$ are identical to the vertices of P [1].

The definition of the straight skeleton can be extended from polygons to planar straight-line graphs (PSLGs) [1], i.e., to collections of straight-line segments that do not intersect except possibly at common end points. The weighted version of the straight skeleton, where edges no longer move at unit speed, was first mentioned by Eppstein and Erickson [9] and studied in detail by Biedl et al. [4].

Eppstein and Erickson's algorithm [9] has $\mathcal{O}(n^{17/11+\epsilon})$ as the currently best known worst-case time and space complexity for unrestricted input with n segments. The algorithm by Vigneron and Yan [27] achieves an expected $\mathcal{O}(n^{4/3}\log n)$ time complexity but is only applicable if no multi-split events occur. Both algorithms are too complex to be implemented, though, and the fastest previously known implementation is by Huber and Held [17].

1.3. Our Contribution

In order to provide a practical tool for mitered offsetting, we converted Aichholzer and Aurenhammer's description [1] of a triangulation-based algorithm into an implementation that can cope with real-world data. In Palfrader et al. [21], we sketched the theoretical basis of an extension and modification 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.

More recently, while implementing our straight-skeleton code SURFER based on the theoretical basis laid out in [21], we investigated the peculiarities of a realization of that algorithm on a standard floating-point arithmetic. For instance, we refine the naive (determinant-based) determination of the collapse times in order to improve the numerical reliability of the algorithm.

In addition to constructing the straight skeleton, SURFER is able to compute mitered offset curves, see Figure 2. SURFER maintains a data structure while computing the straight skeleton which then can be used to quickly compute offset curves for any desired offset by iterating over all faces of the straight skeleton. Note that straight-skeleton based offsetting does not require time-consuming and error-prone operations like computing pairwise intersections or removing excess loops of raw offset curves. Furthermore, offset distances need not be known prior to computing the straight skeleton itself. Having constructed the straight skeleton, one mitered offset curve of a polygon with 100 000 segments can be computed in about 10 ms using a 2010 Intel Core i7-980X CPU clocked at 3.33 GHz.

It is well known that mitered offset intersections for acute angles at reflex vertices can be far away from their defining input. Hence, we extended our approach to support offsetting based on the linear axis [25], resulting in beveled offsets where the distance between any offset curve point and its input is bounded by $\sqrt{2}$ times the orthogonal offset distance. (This distance threshold could also be adapted according to a user-specified parameter.) Figure 3 shows offsets with multi-segment bevels.

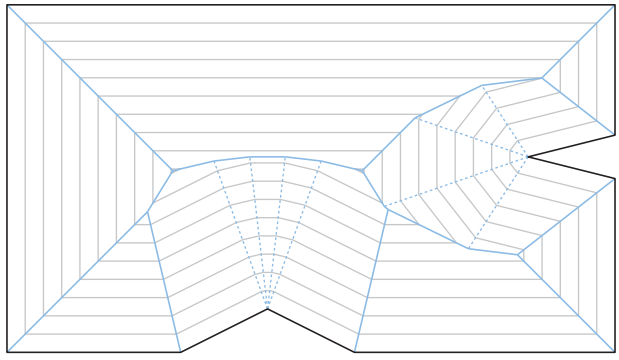

Fig. 3: A linear axis (solid blue) of an input polygon (bold), together with a family of offset curves with multi-segment bevels (gray). The traces of reflex offset vertices are drawn dotted.

We tested SURFER on about twenty thousand polygons and PSLGs, with up to over a million vertices per input. Our tests provide clear evidence that SURFER runs in $\mathcal{O}(n\log n)$ time and linear space for all practical input. More importantly, as witnessed by our tests, it is the fastest (published) algorithm for computing straight skeletons and mitered offsets.

2. TRIANGULATION-BASED ALGORITHM

2.1. The Wavefront Propagation Process

Aichholzer and Aurenhammer's algorithm [1] simulates the propagation of the wavefront to compute the straight skeleton of a polygon or PSLG. To determine when the next event will happen, they use a priority queue of potential future events. The main idea behind their approach is to maintain a kinetic triangulation of the part of the plane that has not yet been reached by the wavefront. (A kinetic triangulation is a triangulation whose defining vertices move.) Every change of the topology of the wavefront and, thus, every edge- or split-event is witnessed by the collapse of a triangle of the kinetic triangulation. Figure 4 demonstrates the process.

Note, however, that not every triangle collapse implies a change in the wavefront: a wavefront vertex may move across a triangulation edge, requiring a local update to the triangulation. This update is called

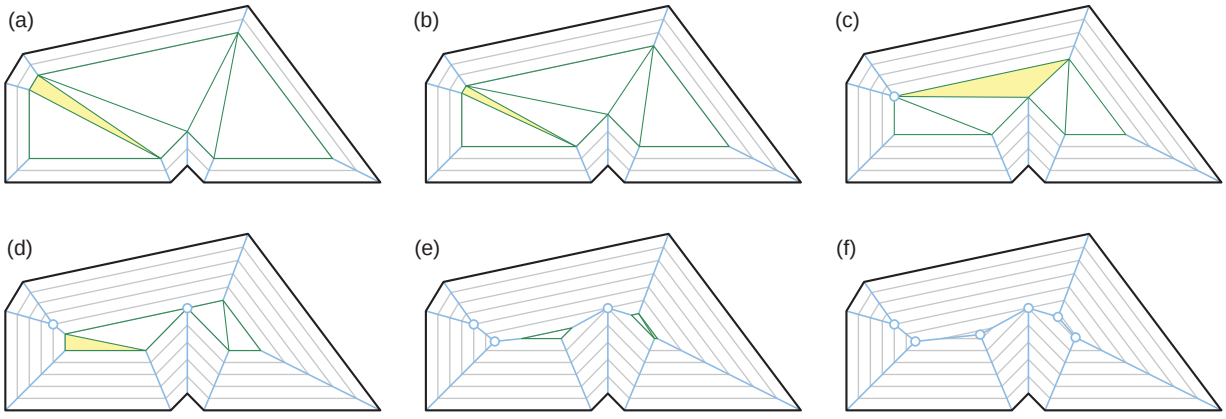

Fig. 4: Wavefront propagation process of a polygon P over time. The kinetic triangulation is shown in green and the parts of the straight skeleton $\mathcal{S}(P)$ that have already been constructed are shown in blue. As any change of the wavefront topology is witnessed by the collapse of a triangle, we have highlighted the next triangle to collapse in every subfigure. In particular, the triangle highlighted in (a) and (b) collapses as its top-left edge shrinks to zero-length in an edge-event, giving rise to the straight-skeleton node first seen in (c). The collapse of the triangle highlighted in (c) witnesses a split event which divides the offset polygon into two, as seen in (d) and (e). Subfigure (f) shows the final straight skeleton and a family of offset curves.

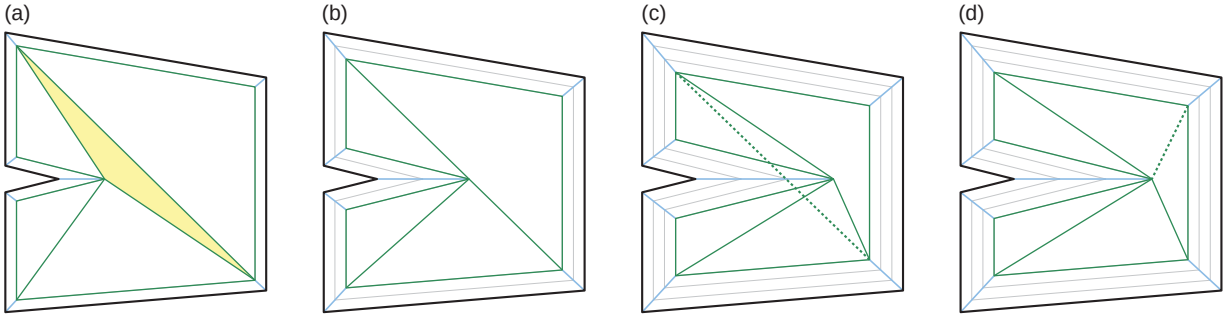

Fig. 5: Not every triangle collapse implies a change of the wavefront-topology, see subfigure (b). We nevertheless have to update the triangulation to stay consistent and avoid a situation like the one shown in (c). By replacing, at the time of collapse, the dotted edge with the other one in the remaining quadrilateral, we retain a valid triangulation of the part of the plane not yet visited by the wavefront, see subfigure (d). This is called a *flip-event*.

a *flip-event* because it consists of flipping a single edge, see Figure 5.

Details of the wavefront propagation process. After constructing the initial wavefront and triangulation, the algorithm computes the collapse times of all triangles and stores them in a priority queue. Then, the wavefront propagation process starts, that is, events are processed from the queue. Each dequeueing operation provides the triangle that collapses soonest. This collapse corresponds to the next event, either an edge- or split-event, or a flip-event. The process ends when the queue is empty.

When the topology of the wavefront changes during an event, some wavefront vertices may merge, in the case of an edge-event, or split, in the case of a split-event. They will also change their velocity when their incident wavefront edges change. Every such change can be seen as old kinetic vertices dying and potentially new kinetic vertices being born to replace some old vertices. This way, the life spans and the speeds of the kinetic vertices may be different, but every vertex moves at constant speed during its specific life span. (A vertex's velocity is a function of the directions and angle between its incident wavefront edges.)

In an edge- or split-event, the triangle collapsed is removed from the kinetic triangulation. Any remaining triangles that are incident to vertices that were replaced need to have their incidences updated and collapse times re-computed as well as their entries in the priority queue updated. In a flip-event, two triangles get replaced by two different triangles as a triangulation edge flips. Again, the priority queue has to be updated accordingly.

After the end of the propagation process, the arcs of the straight skeleton are the line segments traced out by all kinetic vertices over their individual life spans. If we consider these arcs to be directed line segments, then they start either in input vertices or in straight-skeleton nodes, each of which corresponds to one event and, thus, to a triangle collapse. They also

stop in such nodes. If we compute the straight skeleton in the entire plane, then some kinetic vertices may escape to infinity, giving rise to straight-skeleton arcs that are rays rather than line segments.

Collapse times. In order to fill the priority queue, one needs to compute the collapse time of every triangle of the kinetic triangulation. It is well known that the signed area A_Δ of a triangle $\Delta(v_1, v_2, v_3)$ can be obtained by resorting to a determinant computation:

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

where $v_{i,x}$ and $v_{i,y}$ are the x- and y-coordinates of the three vertices v_1, v_2, v_3 of Δ. Since these coordinates are time dependent, the area of a kinetic triangle also is a function of time.

Recall that each kinetic vertex moves at its own, constant velocity. Thus, the coordinates of a triangle's vertices are linear functions of the time t, and the area of a kinetic triangle is a quadratic function in t. Hence, at least in theory, its (up to two) real roots can be computed easily. These roots correspond to collapse times of the triangle and, therefore, to potential event times.

Note that when computing the new collapse times of triangles, either initially at time $t = 0$ or after handling an event at some later time t, only times that are not in the past are considered, that is, only collapse times $t' \geq t$ are valid. If no such solution exists, then the triangle will never collapse and there is no need to insert it into the priority queue. This can happen, for instance, if the straight skeleton of a PSLG in the entire plane or of the outside of a polygon is being constructed and vertices of triangles escape to infinity. It may also happen when computing the interior straight skeleton of a simple polygon, but in this case a later event will cause this triangle's vertices to change and its collapse time to be re-computed.

2.2. Simultaneous Events

The original description [1] of Aichholzer and Aurenhammer's algorithm hinges upon the implicit assumption of general position. In particular, their description lacks any hint on how to proceed if several triangles collapse simultaneously. (Also some proofs of worst-case bounds require all collapse times to be distinct.)

Unfortunately, we observed simultaneous collapses quite frequently for realistic input data. Thus, such an assumption is clearly not warranted for real-world data. Problematic simultaneous collapses primarily manifest themselves in two settings:

1. Parallel lines in the input can cause two or more wavefront edges to move toward one another, causing an area that is covered by many triangles to collapse when the two opposing wavefronts meet. Depending on the exact layout of wavefront edges and vertices, this will result in one or more edge-, split-, and flip-events that all happen at the same time.
2. Several flip-events may happen at the same time between adjacent triangles when their wavefront vertices all become collinear. Processing one of these flip-events may cause one or two of the two new triangles introduced by the flip to also collapse in further flip-events at the very same time. Processing those, in turn, can re-create the original triangles. Thus, this situation may cause the algorithm to get stuck in a *flip-event loop* from which it cannot escape.

Due to inaccuracies caused by finite-precision arithmetic, an implementation might fail to determine that all these events happen at exactly the same time, which complicates correct processing of simultaneous events even more.

We handle problems stemming from the first setting by resorting to the concept of *infinitely fast* moving vertices. Such vertices may be introduced by an event we are handling and then cause any incident triangles to be processed immediately and at the same logical time in the wavefront propagation process.

If the set of all triangles involved in the multi-flip event cloud of the second setting could be assumed to be correct — for instance because the algorithm is run with exact arithmetic — then one can devise an ordering that ensures their eventual resolution [20]. However, with limited precision even establishing that set is impossible. Our solution [21] is to keep a journal of a sufficient number of recent flip events. Should we process the same flip event a second time, we have potentially (though not necessarily) entered a flip-event loop. Once detected, we can apply a special flip-event-loop resolution procedure that identifies the correct set of triangles and re-triangulates the area covered by them and their immediate neighbors.

2.3. Finding and Classifying Collapses

One particular challenge of an implementation of kinetic triangulations is to reliably classify events in order to process collapses correctly and to find these collapse times in the first place: Consider, for instance, a triangle whose three kinetic vertices move such that they meet at the same point at the same time, see Figure 6. The straightforward determinant-based approach to finding its collapse time involves computing the roots of a quadratic polynomial. Since the collapse will happen when all three vertices meet, the resulting quadratic will only touch the x-axis in a single point. That is, the quadratic polynomial has one real root of multiplicity two.

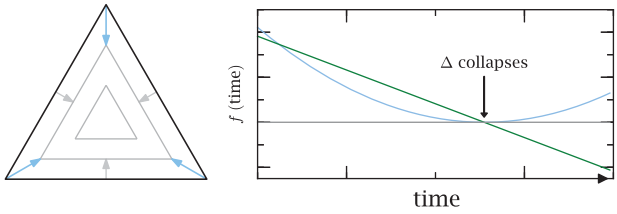

Fig. 6: The collapse time of a triangle (left) can be computed in different ways. The parabola plotted in blue is the (signed) area of the triangle over time. The function in green represents the (signed) distance of one vertex to its opposite edge.

Since we work with finite-precision arithmetic, such as standard IEEE 754 double precision or even the arbitrary-precision library MPFR [11], we have to expect minor numerical inaccuracies to occur in our computations. However, such an inaccuracy might cause us to miss the collapse of that triangle entirely! We mitigate this problem by using alternate means to compute the collapse times whenever possible. As an added bonus, these approaches also help in classifying the events.

Collapsing edges. For finding edge-collapses and collapses which indicate multi-split events, we consider the pairwise distances of each vertex pair of a triangle: Let (v_1, v_2) be a pair of vertices. The distance between them also is a quadratic in time. However, we can use its derivative to find the time t of closest approach between v_1 and v_2. The derivative is linear in time and, hence, more robust to solve. If the line segment $s = (v_1, v_2)$ is a wavefront edge, we know that s must collapse at some point in time (either in the past or future) if we got a unique solution for t — otherwise, the vertices run in parallel and the edge never collapses. If s collapses, so does its incident triangle. If, on the other hand, the line segment $s = (v_1, v_2)$ is not a wavefront edge but a triangulation edge then it need not have collapsed at time t. We compute the length of s at time t, and, if it is reasonably close to zero, we assume a multi-split event is taking place at that time. Otherwise, no event happens.

Vertices moving into edges. For reliably finding the time of split-events and flip-events for triangles that have a wavefront edge we proceed differently. Note that a triangle collapse that indicates a split-event occurs only for triangles with exactly one wavefront edge e. The vertex v opposite of e will be the vertex that crashes into e, causing the split-event. Similarly, when a triangle with a wavefront edge e is involved in a flip-event, it will always be the vertex v opposite of e that is moving across the supporting line $\overline{e}$ of e. Neither split- nor flip-events are possible in triangles with more than one wavefront edge.

To find the time of a potential event, we consider the distance of the vertex v to the edge e over time. We know that e moves at unit speed perpendicular to its direction vector. Furthermore, we can project the velocity of v onto the normal vector of e. The magnitude of this projected vector (either positive or negative, depending on whether v is moving toward or away from e) plus 1 for the edge movement yields the speed of approach. Dividing the orthogonal distance of v to e by the speed of approach yields a possible time for the next split- or flip-event. The actual type of the event depends on whether v intersects $\overline{e}$ in the interior of e itself, or outside of e. If v hits an end point of e then a multi event such as a multi-split event has occurred.

We emphasize that this approach is significantly more robust than the standard determinant-based approach, see again Figure 6.

2.4. Linear Axis

Recall that, when computing the straight skeleton, the initial wavefront is an identical copy of the input polygon. As the wavefront propagation starts, edges move in a self-parallel manner toward the interior of the polygon. This implies that vertices of the wavefront move along the bisectors of input edges at whatever speed is required to "keep up" with the wavefront. Therefore, very acute angles between input edges cause very fast vertices: as an angle approaches 2π, the speed of its vertex approaches infinity. At a reflex input vertex, the distance of its corresponding wavefront vertex to the input thus grows quickly, and the wavefront vertex tends to trace out a long arc. See, for instance, the reflex vertex on the right-hand side of Figure 2 that causes the horizontal arc which spans almost the entire polygon. As a consequence, mitered offsets might be far away from their input.

The linear axis was introduced a decade ago by Tănase and Veltkamp [25] in order to mitigate this problem. Figure 3 shows a linear axis of the same input polygon as previously used for Figure 2. The linear axis is similar to the straight skeleton in that it is defined by an almost identical wavefront propagation process. In particular, it also consists of traces of wavefront vertices over a propagation process. It differs from the straight skeleton in the handling of reflex vertices.

Handling of reflex vertices. For a linear axis, more than one wavefront vertex may emanate from a reflex input vertex. This is achieved by adding zero-length wavefront edges, so-called *hidden edges*, to the initial wavefront at reflex vertices. The directions of these edges are chosen such that the interior angle between any pair of adjacent wavefront edges sent out from one reflex input vertex is the same. This results in the wavefront vertices fanning out uniformly from each reflex vertex. Furthermore, this choice bounds the speed of any wavefront vertex. The particular bound depends on the exact number of vertices sent forth,

but it is always less than or equal to $\sqrt{2}$, approaching 1 as the number of hidden edges goes to infinity. Recall that edges move at unit speed. Therefore, in any given offset curve of orthogonal offset distance d, no vertex is farther than $\sqrt{2} \cdot d$ from any input.

Traces of reflex vertices. The second difference is merely technical in nature. While the straight skeleton consists of traces of all wavefront vertices over the propagation process, the linear axis excludes traces of wavefront vertices emanated from reflex vertices. The dotted segments in Figure 3 correspond to the arcs dropped from the linear axis; the linear axis itself consists only of the solid, blue lines. Tănase and Veltkamp [25] prove that the linear axis converges to the medial axis as the number of wavefront vertices sent out at every reflex vertex goes to infinity. (Compare the solid blue structures of Figure 1 and Figure 3.)

Computing the linear axis. We extended our straight-skeleton algorithm to support computing the linear axis. Currently, the number of hidden edges at each reflex vertex is the same everywhere, but our code could easily be adapted to make this a per-vertex property, possibly based on the angle between its incident input edges, or on per-vertex input provided by the user.

3. OFFSETTING BASED ON STRAIGHT SKELETONS

A straightforward approach to straight-skeleton based offsetting is to simply dump the wavefront when the desired offset distance t is reached. If another offset curve for an offset distance $t' > t$ is sought afterwards, we can simply continue the wavefront propagation, starting at t and proceeding until t' is reached. This approach to offsetting is supported by our implementation, SURFER.

However, if multiple offsets were to be computed for offset distances that are not arranged in increasing order, we would have to re-start the wavefront-propagation process at $t = 0$ for each new offset distance t' that is smaller than the previous distance t. Note that an outward wavefront propagation that starts at $\mathcal{W}_P(t)$ will not, in general, restore $\mathcal{W}_P(t')$. (Rather, one would have to store the entire wavefront-propagation history, including all topological changes.)

If multiple offsets are sought in a non-sorted order, it seems more natural to compute the full straight skeleton (or linear axis) in a preprocessing step, and to exploit the information embedded in the straight skeleton to quickly and efficiently compute offset curves. (We show experimentally that this approach is competitive even if just a single offset is sought, see Section 4.)

A mitered offset curve of a polygon or PSLG consists of straight-line segments only. These line segments are grouped within one or more polygons. (This is true even for PSLGs that do not form closed polygons.) The straight skeleton partitions a polygon or the plane into faces, similarly to the partition induced by a Voronoi diagram. Each face is bounded on one side by the input edge that swept this face. On the other sides, a face is bounded by straight-skeleton arcs. Offset curve segments in a given face are always parallel to the input edge that defined this face, and the distance between the offset segment and the supporting line of the input edge is the offsetting distance. Thus, line segments that make up an offset curve never have their end points within a face. Instead, end points always lie on the boundary of faces, i.e., on straight-skeleton arcs. See the offset curves in Figure 2 for an example.

Of course, similar considerations also apply to the linear axis if one includes all arcs traced out by reflex vertices. (That is, the blue dotted segments in Figure 3.) For some faces incident at reflex vertices, the defining input edges will then be the hidden edges discussed in Section 2.4. Using the linear axis will result in beveled offsets as seen in Figure 3. The number of hidden edges at a reflex vertex corresponds to the number of bevel segments at that corner.

3.1. Computing One Mitered Offset

Recall that arcs in the straight skeleton are those line segments which have been traced out by kinetic vertices in the propagation process. Thus, there is a one-to-one correspondence between kinetic vertices and straight-skeleton arcs, and our implementation just needs to keep track of kinetic vertices, including their start and end times and loci. These start and end times also induce a natural orientation on the straight-skeleton arcs.

While computing the straight skeleton, we maintain a data structure that represents the straight-skeleton faces: For each input segment, we store a reference to its two incident kinetic vertices. Each kinetic vertex has pointers to the next (clockwise) and previous (counterclockwise) kinetic vertex for both of its incident faces. As a result, we are able to traverse the boundary of every straight-skeleton face.

Now recall that the time span of a kinetic vertex has a one-to-one correspondence to the range of (orthogonal) offset distances traveled by the vertex. Thus, an arc a of the straight skeleton is intersected by an offset curve for offset distance d if and only if d lies within the time span of the kinetic vertex that defined a. We call this time span the *offset interval* of a.

To compute all offset curves for an offset distance d we proceed as follows: We scan all straight-skeleton arcs until we find a so-far unvisited arc a whose offset interval contains d. No particular order of arcs is

```
Algorithm offset(d, A):
    Input: Offsetting distance d, set of arcs A
    offset ← [ ] ;
    for a ∈ A do
        if ¬ a.seen ∧ a.is_alive_at(d) then
            offset.append(one_curve(d, a)) ;
        end
    end
    return offset ;
end
```

```
Algorithm one_curve(d, a):
    Input: Offsetting distance d, arc a alive at d
    curve ← [ ],  start ← a;
    repeat
        curve.append(a.position_at_time(d)) ;
        a.seen ← true;
        face ← a.right_face;
        repeat
            a ← a.next_cw_in_face(face) ;
        until a.is_alive_at(d);
    until a = start;
    return curve ;
end
```

Fig. 7: Computing an offset once the straight skeleton is known.

required for this scan as long as we make sure that all arcs are scanned eventually and that no arc is scanned repeatedly.

Let f be the face to the right of a, and let s be input segment that swept f. Furthermore, let v be the kinetic vertex that traced out a. To obtain the first vertex of an offset curve, we compute the locus of v at time d. This locus corresponds to the intersection of a with an offset segment of s at distance d. Then, we advance clockwise along the boundary of f until we encounter another straight-skeleton arc a' whose offset interval contains d. We compute the offset vertex on a' and leave f to move to the face on the other side of a'.

We repeat this step until we return to the initial arc a, marking all straight-skeleton arcs that were encountered as visited. Meeting a again indicates that one offset polygon has been fully determined. A continued scan of the remaining unmarked straight-skeleton arcs reveals all further offset polygons for the offset distance d. This algorithm is sketched in Figure 7.

Thus, straight-skeleton based offsetting is as simple as traversing a planar graph. Obviously, the same process can be used to compute beveled offsets based on the linear axis, provided that the dotted arcs of Figure 3 are added to the linear axis.

Since the number of straight-skeleton arcs (kinetic vertices) is linear in the number n of segments of the input PSLG, and because the amount of work we have to do is constant per arc, this offsetting procedure runs in time linear in n, i.e., in $\mathcal{O}(n)$ time. In practice, generating one offset curve is extremely fast if the straight skeleton is already known, taking a few milliseconds even for input on the order of hundreds of thousands of vertices, see Figure 10b in Section 4.

3.2. Comparison with Other Approaches

Known straight-skeleton codes include the implementations by Felkel and Obdržálek [10], Cacciola [6], and Huber and Held [16]. Cacciola's code is a corrected and improved version of the Felkel-Obdržálek algorithm which is known to be flawed [15,28]; it is shipped with the CGAL library [7]. These codes use similar data structures for offsetting and, thus, also generate the same kind of mitered offsets. (They differ only in speed, see Section 4.)

Interestingly, even a thorough literature and Web search reveals hardly any other published (and non-proprietary) implementation that supports mitered offsetting. It is easy to find several Java applets that compute mitered offsets, but their authors frankly admit that these codes are either not reliable or cannot handle general polygons and arbitrary offset distances.

The only notable exceptions are the well-known polygon-clipping libraries CLIPPER [18] and GEOS [23]. Both libraries apply general-purpose Boolean clipping algorithms to compute mitered and (approximations of) rounded offsets.

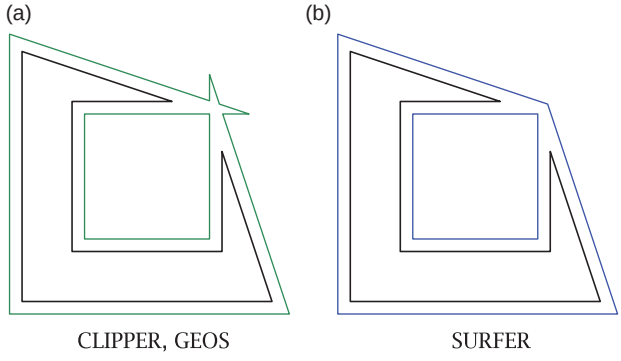

Fig. 8: An input polygon (bold) with one offset curve. Note that the offsets differ.

Note, though, that their offsets are different from those computed by means of straight skeletons, see Figure 8. Without knowing details of the offsetting algorithm employed by these codes, it is impossible to pin down the exact reason why the two excess areas in Figure 8a are not removed. Both libraries seem to compute winding numbers [8] in order to weed out incorrect regions within the union of elementary offset areas that form the raw offset polygons.

Our tests suggest that the same winding-number algorithm is applied for both rounded and mitered offsets. However, the rounded-offsetting problem and the mitered-offsetting problem differ in spirit since mitered offsetting allows vertices to travel arbitrary distances. Straight-skeleton based offsetting prevents portions of the wavefront to penetrate each other, resulting in different offsets. But, of course, applications might exist that prefer CLIPPER's or GEOS's offsets over straight-skeleton based offsets.

4. EXPERIMENTAL RESULTS

What is the price that we have to pay for computing the full straight-skeleton as a preprocessing step? Is this approach competitive with known approaches to mitered offsetting? In order to answer these questions, we implemented our algorithm in C. Our implementation, SURFER, can work with standard IEEE 754 double-precision floating-point arithmetic as well as with arbitrary-precision arithmetic based on the MPFR library [11].

We conducted performance tests with SURFER on about twenty thousand polygons and PSLGs with up to over a million vertices per input, consisting of both real-world and contrived data of different characteristics, including CAD/CAM designs, printed-circuit board layouts, geographic maps (e.g., Figure 9), fractal and space filling curves (e.g., Figure 11), star-shaped polygons and random polygons generated by RPG [3], as well as sampled spline curves, families of offset curves and font outlines. Some datasets contain circular arcs, which we approximated by polygonal chains in a preprocessing step. See our web-page [12] for high-resolution images of straight skeletons and mitered offsets computed by SURFER.

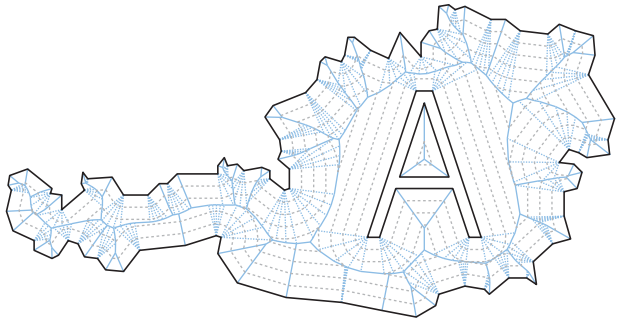

Fig. 9: A simplified outline of Austria with hole (solid black), a linear axis (solid blue) with the traces of reflex vertices (dotted blue), and a family of offset curves (dashed gray).

Previous tests [21] showed that SURFER runs in $\mathcal{O}(n \log n)$ time and linear space for practically all n-segment inputs. (From a theoretical point of view, one could design contrived inputs that cause the algorithm to consume more time, but we have not encountered any such input in our tests.) In particular, SURFER is faster than BONE, the fastest known implementation of a straight-skeleton algorithm by Huber and Held [16], which in turn is significantly faster than Cacciola's implementation [6] that is shipped with the CGAL library [7]: As reported previously [21], extensive experiments suggest an average runtime (in seconds) of $5.8 \cdot 10^{-7} n \log n$ for SURFER for an n-segment input, $1.5 \cdot 10^{-5} n \log n$ for BONE, and $4.5 \cdot 10^{-7} n^2 \log n$ for Cacciola's CGAL code.

4.1. Computing a Single Offset

In the following, we compare the running time of SURFER in its IEEE 754 double-precision mode against the C++ version of CLIPPER. We should stress that mitered offsetting is not the primary goal of either CLIPPER or GEOS, but both libraries are frequently recommended as offsetting engines on the Web.

For this test, we limited our inputs to simple polygons (without holes) and computed one interior offset. Since CLIPPER accepts only integer coordinates we generated input for CLIPPER by scaling every polygon to fit into the unit square, multiplying every vertex coordinate by 10^6, and then casting it to an integer. In order to allow an automated run of the tests, for every test polygon, we set the offset distance to one quarter of the radius of the maximum inscribed circle of the polygon. All offsetting computations were terminated if they exceeded more than ten minutes of running time. The wavefront propagation of SURFER was stopped once the required offset distance was reached, and then the procedure outlined in Section 3.1 was used to construct the offset curve.

The results are plotted in Figure 10a. (All tests were conducted on a 2010 Intel Core i7-980X CPU clocked at 3.33 GHz.) This plot clearly shows that SURFER tends to be faster than CLIPPER for virtually all data sets. Our tests revealed that the difference in speed becomes more pronounced as input gets more complex or offsetting distances become larger. Also, the variation in the running times is much larger for CLIPPER than for SURFER. In any case, the plot suggests that CLIPPER's offsetting suffers from a quadratic (or even worse) average-case complexity. Results for the GEOS library are not plotted since it tends to be significantly slower than CLIPPER, running about 100 times longer than CLIPPER.

Our tests also demonstrated that both SURFER and CLIPPER require a linear amount of memory. SURFER needs roughly 1 MB of memory per 1 000 input vertices. Hence, for an input of 100 000 vertices SURFER requires approximately 100 MB of RAM. This is about twice the footprint of CLIPPER.

4.2. Computing Many Offsets

The plot of Figure 10b shows the time consumed by SURFER for computing the full straight skeleton and

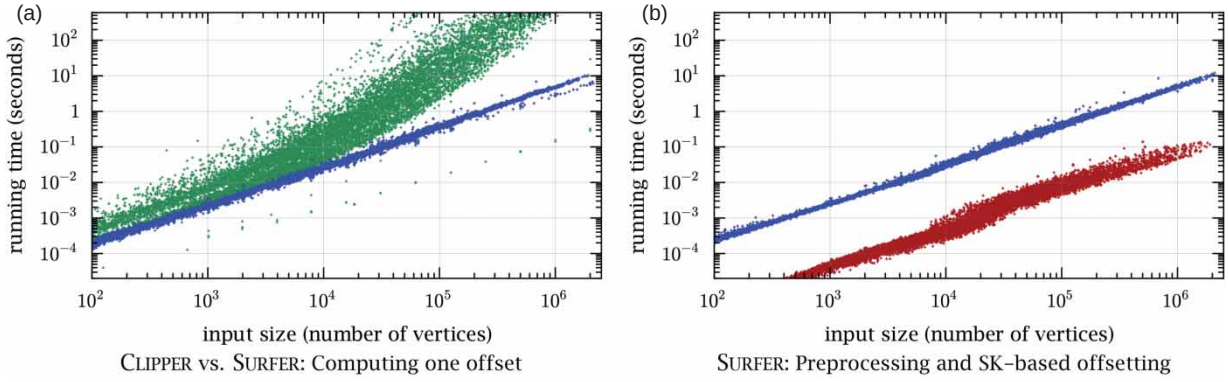

Fig. 10: (a) Run-time comparison between CLIPPER 6.1.3 (green) and SURFER (blue) when computing interior offsets. (b) Time required by SURFER to compute the full straight skeleton (blue) and a single offset once the straight skeleton is known (red).

for generating one offset once the straight skeleton is available. In order to get reliable timings even for small inputs, we averaged the running times of the offsetting step over at least one thousand computations per input. (Once the straight skeleton is known, the running time of an offset computation does not depend on the actual offset distance chosen.)

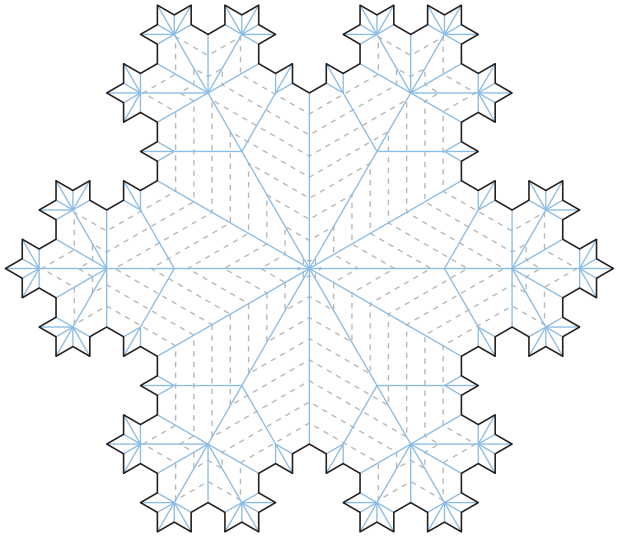

Fig. 11: A Koch snowflake (solid black), its straight skeleton (solid blue), and a family of corresponding offset curves (dashed gray).

5. CONCLUSION

We explain how the triangulation-based straight-skeleton algorithm by Aichholzer and Aurenhammer [1] can be extended to make it compute (1) straight skeletons, (2) linear axes, and (3) mitered and beveled offsets of arbitrarily complex real-world PSLGs on a finite-precision arithmetic. We implemented both the straight-skeleton algorithm and the offsetting algorithm in a C library called SURFER.

Extensive tests clearly demonstrate that the resulting implementation is the fastest code for mitered offsetting currently available: offsetting can be performed on a standard PC within a few milliseconds even for a fairly complex PSLG with several hundreds of thousands of segments if the straight skeleton of the PSLG has been computed in a preprocessing step. Since SURFER consumes only about 60 µs to 70 µs per input segment, the computation of the straight skeleton seems to be well warranted even if only few offset curves have to be generated. In particular, computing even just one offset by SURFER based on straight skeletons tends to be significantly faster than offsetting by means of CLIPPER, even if the time consumed by SURFER's straight-skeleton computation is included in the timings.

An obvious task for future work is the extension of mitered offsetting to offsets with non-uniform offsetting distances. One could, for instance, fix an offset distance d for one input edge and then specify all other offset distances as fractions or multiples of d. Such a non-uniform offsetting could be realized based on weighted straight skeletons. Baring some technical details which still are to be resolved, we believe that SURFER could be extended to handle also this more general offsetting problem.

ACKNOWLEDGEMENTS

This work was supported by Austrian Science Fund (FWF): P25816-N15.

REFERENCES

[1] Aichholzer, O.; Aurenhammer, F.: Straight Skeletons for General Polygonal Figures in the Plane. A. Samoilenko (Editor), Voronoi's Impact on Modern Science, Book 2. Inst. Mathematics National Academy Sci. Ukraine, 1998, 7–21. doi:10.1007/3-540-61332-3_144.

[2] Aichholzer, O.; Aurenhammer, F.; Alberts, D.; Gärtner, B.: A Novel Type of Skeleton for Polygons. J. Universal Comput. Sci., 1(12), 1995, 752–761. doi:10.1007/3-540-61332-3_144.

[3] Auer, T.; Held, M.: Heuristics for the Generation of Random Polygons. Proc. Canad. Conf. Comput. Geom. (CCCG'96). Carleton University Press, Aug 1996, 38–44.

[4] Biedl, T.; Held, M.; Huber, S.: Reconstructing Polygons from Embedded Straight Skeletons. Proc. 29th Europ. Workshop Comput. Geom. Mar 2013, 95–98.

[5] Blum, H.: A Model for Extracting New Descriptors of Shape. W. Dunn (Editor), Models for the Perception of Speech and Visual Form. MIT Press, 1967, 289–310.

[6] Cacciola, F.: 2D Straight Skeleton and Polygon Offsetting. CGAL User and Reference Manual. CGAL Editorial Board, 4.3 edition, 2013.

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

[8] Chen, X.; McCains, S.: Polygon Offsetting by Computing Winding Numbers. Proc. ASME 2005 Int. Design Engg. Techn. Conf. & Computers Inf. Engg. Conf. (IDECT/CIE'05). Sep 2005, 565–575.

[9] Eppstein, D.; Erickson, J.: Raising Roofs, Crashing Cycles, and Playing Pool: Applications of a Data Structure for Finding Pairwise Interactions. Discrete Comput. Geom., 22(4), 1999, 569–592. doi:10.1145/276884.276891.

[10] Felkel, P.; Obdržálek, Š.: Straight Skeleton Implementation. Proc. 14th Spring Conf. Comput. Graphics. 1998, 210–218.

[11] Fousse, L.; Hanrot, G.; Lefèvre, V.; Pélissier, P.; Zimmermann, P.: MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. ACM Trans. Math. Software, 33(2), June 2007, 13:1–13:15. doi:10.1145/1236463.1236468. http://www.mpfr.org/.

[12] Held, M.: Weighted and Unweighted Straight Skeletons. http://www.cosy.sbg.ac.at/~held/projects/wsk/wsk.html.

[13] Held, M.: On the Computational Geometry of Pocket Machining, *Lecture Notes Comput. Sci.*, volume 500. Springer-Verlag, June 1991. ISBN 3-540-54103-9.

[14] Held, M.: VRONI and ArcVRONI: Software for and Applications of Voronoi Diagrams in Science and Engineering. Proc. 8th Int. Symp. Voronoi Diagrams in Science & Engineering. June 2011, 3–12. doi:10.1109/ISVD.2011.9.

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

[16] Huber, S.; Held, M.: Theoretical and Practical Results on Straight Skeletons of Planar Straight-Line Graphs. Proc. 27th Annu. ACM Sympos. Comput. Geom. June 2011, 171–178.

[17] Huber, S.; Held, M.: A Fast Straight-Skeleton Algorithm Based on Generalized Motorcycle Graphs. Internat. J. Comput. Geom. Appl., 22(5), Oct 2012, 471–498. doi:10.1142/S0218195912500124.

[18] Johnson, A.: Clipper — An Open Source Freeware Library for Clipping and Offsetting Lines and Polygons, Jan 2014. http://www.angusj.com/delphi/clipper.php.

[19] Li, M.; Zhang, L.-C.; Mo, J.-H.; Zhao, Z.-Y.: A Unified Method for Invalid 2D Loop Removal in Tool-Path Generation. Comput. Aided Design, 45(2), Feb 2013, 124–133. doi:10.1016/j.cad.2012.07.010.

[20] Palfrader, P.: Computing Straight Skeletons by Means of Kinetic Triangulations. MSc thesis, Univ. Salzburg, CS Dept., Salzburg, Austria, Sep 2013. http://www.palfrader.org/research/2013/master-thesis.pdf.

[21] Palfrader, P.; Held, M.; Huber, S.: On Computing Straight Skeletons by Means of Kinetic Triangulations. Proc. 20th Annu. Europ. Symp. Algorithms (ESA'12). Sep 2012, 766–777. doi:10.1007/978-3-642-33090-2_66.

[22] Persson, H.: NC Machining of Arbitrarily Shaped Pockets. Comput. Aided Design, 10(3), May 1978, 169–174. doi:10.1016/0010-4485(78)90141-0.

[23] Santilli, S.; et al.: GEOS - Geometry Engine, Open Source, Aug 2013. http://trac.osgeo.org/geos/.

[24] Sugihara, K.: Design of Pop-Up Cards Based on Weighted Straight Skeletons. Proc. 10th Int. Symp. Voronoi Diagrams in Science & Engineering (ISVD'13). July 2013, 23–28. doi:10.1109/ISVD.2013.9.

[25] Tănase, M.; Veltkamp, R. C.: A Straight Skeleton Approximating the Medial Axis. Proc. 12th Annu. Europ. Symp. Algorithms. Sep 2004, 809–821. doi:10.1007/978-3-540-30140-0_71.

[26] Tomoeda, A.; Sugihara, K.: Computational Creation of a New Illusionary Solid Sign. Proc. 9th Int. Symp. Voronoi Diagrams in Science & Engineering (ISVD'12). June 2012, 144–147. doi:10.1109/ISVD.2012.26.

[27] Vigneron, A.; Yan, L.: A Faster Algorithm for Computing Motorcycle Graphs. Proc. 29th Annu. ACM Sympos. Comput. Geom. 2013, 17–26. doi:10.1145/2462356.2462396.

[28] Yakersberg, E.: Morphing Between Geometric Shapes Using Straight-Skeleton-Based Interpolation. MSc thesis, CS Dept., Technion, Haifa, Israel, May 2004.