# An Intersection-Sensitive Algorithm for Snap Rounding

Mark de Berg* Dan Halperin† Mark Overmars‡

**Abstract**

Snap rounding is a method for converting arbitrary-precision arrangements of segments into fixed-precision representation. We present an algorithm for snap rounding with running time $O((n + I) \log n)$, where $I$ is the number of intersections between the input segments. In the worst case, our algorithm is an order of magnitude more efficient than the best previously known algorithms. We also propose a variant of the traditional snap-rounding scheme. The new method has all the desirable properties of traditional snap rounding and, in addition, guarantees that the rounded arrangement does not have degree-2 vertices in the interior of edges. This simplified rounded arrangement can also be computed in $O((n + I) \log n)$ time.

## 1 Introduction

Robustness and precision issues are major stumbling blocks to successful implementation of geometric algorithms. Tremendous effort has been exerted over the years to overcome these problems; see the recent surveys on the topic by Schirra [14] and by Yap [17]. It is typically assumed in the theoretical study of geometric algorithms that we have an infinite-precision real arithmetic machine at our disposal (the so-called real RAM [13]) and that the input is degeneracy free. Of course, these assumptions are not realistic and in practice require extra work to relax.

The approaches taken to solve the robustness problems can be roughly categorized in two groups. The first group of solutions mimics the real-RAM machine by supplying special arithmetic that is exact for a limited set of objects [10, 16]. The second group makes do with limited precision arithmetic and adapts the algorithms to give meaningful and useful results in spite of this precision restriction [7, 9, 11, 15].

*Department of Computer Science, TU Eindhoven, PO Box 513, 5600 MB Eindhoven, the Netherlands. MdB was supported by NWO under project no. 639.023.301.

†School of Computer Science, Tel Aviv University, Tel Aviv 67789, Israel. Work by DH reported in this paper has been supported in part by the IST Programme of the EU as Shared-cost RTD (FET Open) Project under Contract No IST-2001-39250 (MOVIE - Motion Planning in Virtual Environments), by The Israel Science Foundation founded by the Israel Academy of Sciences and Humanities (Center for Geometric Computing and its Applications), and by the Hermann Minkowski – Minerva Center for Geometry at Tel Aviv University.

‡Department of Information and Computing Sciences, Utrecht University, P.O.Bos 80.089, 3508 TB Utrecht, the Netherlands.

Geometric rounding, which is the topic of our paper, has the goal of transforming an arbitrary precision geometric object into a finite-precision representation. While it is a fixed-precision scheme, it is also relevant when we use exact computing. Quite often, the results (coordinates) of exact geometric computing require prohibitively large number size (number of bits) or simply cannot be expressed numerically with a finite number of bits (when algebraic numbers are involved), making it difficult to further manipulate the results in applications.

Consistent rounding is therefore a major goal in geometric computing in practice. So far it has been achieved in only a limited number of instances, most notably by the elegant scheme *snap rounding* for arrangements of segments, originally proposed in [5] and [9]. Snap rounding works as follows.

Let $\mathcal{S}$ be a finite collection of line segments in the plane. The arrangement $\mathcal{A}(\mathcal{S})$ of $\mathcal{S}$ is the decomposition of the plane into vertices, edges, and faces induced by $\mathcal{S}$. Given a collection of segments whose endpoints are represented with arbitrary-precision coordinates, snap rounding proceeds as follows [5, 9]. We tile the plane with a grid of unit squares, *pixels*, each centered at a point with integer coordinates. A pixel is *hot* if it contains a vertex of the arrangement. Each vertex of the arrangement (which is either a segment endpoint or an intersection point of two segments) is replaced by the center of the hot pixel containing it and each edge $e$ is replaced by the polygonal chain through the centers of the hot pixels met by $e$, in the same order as they are met by $e$. Note that in the process, vertices, edges, and faces of the original arrangement may collapse. Guibas and Marimont [6] proved that the snap-rounded arrangement $\mathcal{A}^*$ has the following desirable properties:

***Fixed-precision representation:*** All vertices of $\mathcal{A}^*$ are at centers of grid squares. (This is not completely trivial, as one has to argue that the rounding procedure does not introduce new intersections.)

***Geometric similarity:*** For each $s \in \mathcal{S}$, the approximation $s^*$ lies within the Minkowski sum of $s$ and a pixel centered at the origin.

***Topological similarity:*** $\mathcal{A}$ and $\mathcal{A}^*$ are "topologically equivalent up to the collapsing of features". More precisely, there is a continuous deformation of the segments in $\mathcal{S}$ to their snap-rounded counterparts such that no segment ever passes completely over a vertex of the arrangement.

The most efficient algorithm to compute the snap-rounded arrangement for a given set $\mathcal{S}$ of segment is by Goodrich et al. [3]. The running time of the algorithm is $O(n \log n + \Sigma_{h \in H} |S_h| \log n)$, where $H$ denotes the set of all hot pixels and $\mathcal{S}_h$ denotes the collection of segments intersecting a hot pixel $h$. The complexity of a rounded arrangement is at most $O(n^2)$. Unfortunately, as Halperin and Packer [8] observed, $\Sigma_{h \in H} |S_h|$ can be as much as $\Theta(n^3)$—see Fig. 1. Hence, the worst-case running time of the algorithm of Goodrich et al. is $\Theta(n^3 \log n)$. The first contribution of our paper is an algorithm that is much better in the worst-case; it runs in time $O((n + I) \log n)$, where $I$ denotes the number of intersections points in $\mathcal{A}(\mathcal{S})$, which is $\Theta(n^2 \log n)$ in the worst case.

Although a snap-rounded arrangement has several nice properties, as mentioned above, it is not completely satisfactory. For instance, the distance between a vertex and a non-incident edge

Figure 1: An arrangement where $\Sigma_{h \in H} |S_h| = \Theta(n^3)$ [8].

in a snap-rounded rounded arrangement can still be arbitrarily small. This means the use of finite-precision arithmetic may still cause problems. Hence, Halperin and Packer [8] introduced *iterated snap rounding*, where the rounding process is repeated—that is, edges that come too close to a vertex after rounding are snapped to that vertex—until there is a minimum separation between vertices and edges. Unfortunately, the drift of a segment in this scheme can be very large [12].

Another unpleasant side-effect of snap rounding is that it can introduce degree-2 vertices that are not segment endpoints—see Fig. 2. This is an unwanted situation. Hence, we formulate a

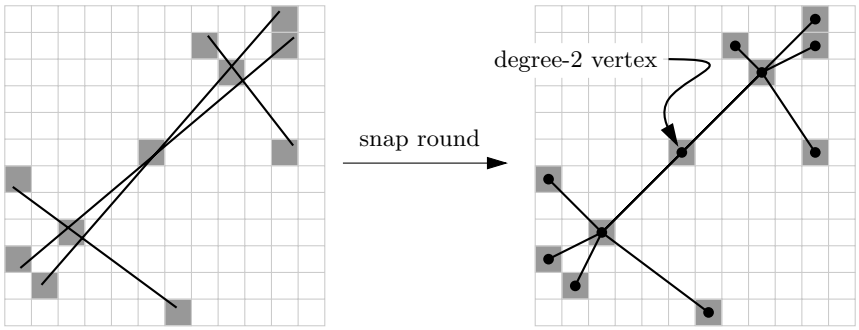


Figure 2: Snap rounding can induce degree-2 vertices.

fourth desirable property:

***Non-redundancy:*** Any degree-2 vertex in $\mathcal{A}^*$ corresponds to an endpoint of an original segment.

The second contribution of our paper is to show that one can satisfy the three traditional properties—fixed-precision representation, geometric similarity, and topological similarity—and, in addition, the non-redundancy property. In particular, we prove that after simply deleting the degree-2 vertices not corresponding to segment endpoints—that is, replacing its two incident edges by a single edge—we still have the other properties. This also implies that we can compute a snap-rounded arrangement that has all four properties in $O((n + I) \log n)$ time.

# 2 An intersection-sensitive algorithm

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ be the collection of input segments whose arrangement $\mathcal{A}(\mathcal{S})$ we wish to round. Following Guibas and Marimont [6], we call these segments *ursegments*. To simplify the presentation, we assume that we have chosen the grid such that no ursegment is horizontal

or vertical; if necessary, our algorithm can easily be adapted to handle horizontal and vertical ursegments. (Note that the rounded arrangement can still have horizontal and vertical edges.) We present an algorithm for snap rounding that is sensitive to the complexity of $\mathcal{A}(\mathcal{S})$, rather than to the overall complexity of the chains representing the ursegments. The output of the algorithm is a graph $\mathcal{G} = (V, E)$ whose nodes are in one-to-one correspondence with the hot pixels, and that has an arc between any two nodes $v_1, v_2 \in V$ for which there is at least one ursegment crossing both corresponding hot pixels $h_1$ and $h_2$ and no other hot pixels in between. The desired rounded arrangement is the planar straight-line embedding of this graph, where each vertex is located at the center of its corresponding hot pixel.

The algorithm starts with computing the set $H$ of hot pixels by finding all the vertices of the arrangement $\mathcal{A}(\mathcal{S})$. This gives us the nodes of the graph $\mathcal{G}$. It remains to find the arcs, which we do in two stages. In the first stage, which is detailed below, we find the arcs in $\mathcal{G}$ for which there is an ursegment with positive slope connecting the hot pixels that are the endpoints of the arc. In the second stage we find the arcs for which there is a connecting ursegment with negative slope; since it is symmetric to the first stage, we omit its description. Note that horizontal arcs may be found twice, but this does not influence the asymptotic running time of the algorithm.

Let $\mathcal{S}^+$ be the subset of ursegments with positive slope. We will find the arcs in $\mathcal{G}$ induced by $\mathcal{S}^+$ with a sweep-line algorithm. More precisely, our algorithm sweeps over the plane with a polygonal curve, moving over the pixels in lexicographical order: pixels are treated from left to right, and within a column pixels are treated from bottom to top. Inside a pixel we sweep from left to right. Hence, the sweep line $\ell$ consists of up to five links, which are horizontal or vertical—see Fig. 3. Note that any ursegment in $\mathcal{S}^+$ either does not intersect the sweep line, or intersects the sweep line in a single point.

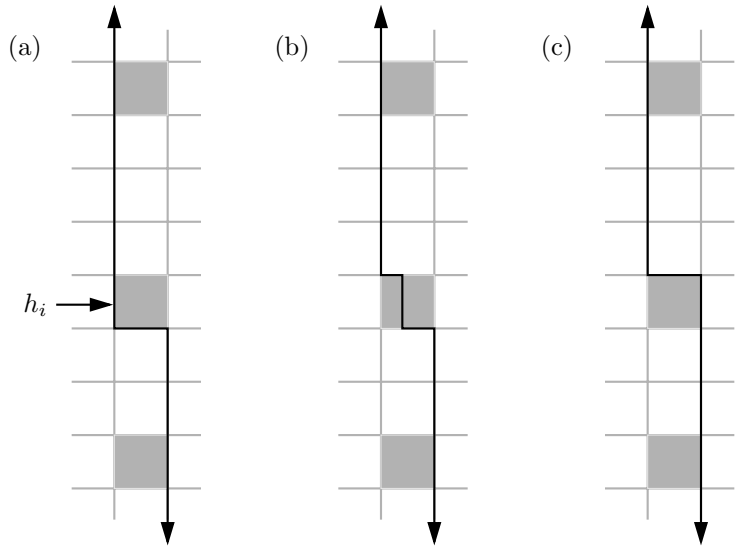


Figure 3: Sweeping over a pixel.

The goal of our algorithm is to report an arc connecting two hot pixels only once, even when there are many ursegments connecting the hot pixels; this is the key to our algorithm's efficiency.

To this end we group ursegments into so-called bundles. We orient each ursegment from left to right. A *bundle* $b = (\mathcal{S}_b, h_b)$ is a collection $\mathcal{S}_b \subset S^+$ of ursegments all coming out of the same hot pixel $h_b$. A bundle persists as long as it does not cross another hot pixel, where we say that a bundle $(\mathcal{S}_b, h_b)$ *crosses* a hot pixel $h \neq h_b$ if either (i) an ursegment of $\mathcal{S}_b$ crosses $h$, or (ii) when there is a vertical line that crosses two ursegments of $\mathcal{S}_b$ and a hot pixel $h$ between them (even if no ursegment of $\mathcal{S}_b$ crosses $h$).

Since a bundle $b = (\mathcal{S}_b, h_b)$ persists from the hot pixel $h_b$ until it crosses another hot pixel, the vertical order of the ursegments in $\mathcal{S}_b$ does not change. We denote the uppermost ursegment of the bundle by $u(b)$ and the lowest ursegment of the bundle by $l(b)$.

Our algorithm sweeps a "vertical" polygonal sweep line $\ell$ from left to right, as explained above and illustrated in Fig. 3, stopping to handle events at hot pixels. In between two events, the bundles are disjoint along $\ell$. We maintain the bundles in the following status structures:

- We have a search tree $\mathcal{T}$ that stores for each bundle $b$ intersecting the sweep line $\ell$ the delimiting ursegments $u(b)$ and $l(b)$, in the order in which they intersect $\ell$. (Notice that since the bundles are disjoint, $u(b)$ and $l(b)$ are adjacent in $\mathcal{T}$ for every bundle $b$ that intersects $\ell$.)
- For each bundle $b = (\mathcal{S}_b, h_b)$, we have a search tree $\mathcal{T}_b$ that stores the ursegments in $\mathcal{S}_b$ intersecting $\ell$ in the order in which they intersect $\ell$, and that allows for splitting and merging in $O(\log n)$ time. E.g. a red-black tree [4] can be used for this.

We also remember for each bundle from which hot pixel it originated. We are now ready to describe the actions taken when the sweep line reaches a hot pixel $h$.

*Step 1: Find all bundles crossing $h$.*

Let $se(h)$ and $nw(h)$ be the lower right and top left corner of $h$. These two corners define an interval on $\ell$, which we denote by $[se(h) : nw(h)]$. The two delimiting segments $l(b)$ and $u(b)$ of any bundle $b$ also define an interval on $\ell$, denoted $[l(b) : u(b)]$. If $b$ crosses $h$, then either $l(b) \cap \ell \in [se(h) : nw(h)]$, or $u(b) \cap \ell \in [se(h) : nw(h)]$, or $[se(h) : nw(h)] \subset [l(b) : u(b)]$. Bundles of the first two types can be found by searching with the interval $[se(h) : nw(h)]$ in $\mathcal{T}$. This takes time $O(\log n + k)$, where $k$ is the number of reported bundles. Because the bundles are disjoint along $\ell$ when $\ell$ reaches $h$, there can be at most one bundle of the third type (in fact, there can only be such a bundle if there are no bundles of the first two types). This bundle, if it exists, can be found using $\mathcal{T}$ in $O(\log n)$ time.

*Step 2: Split the bundles and create new bundles.*

For each bundle $(\mathcal{S}_b, h_b)$ found in step 1, we split $\mathcal{S}_b$ into three subsets: (i) the set $\mathcal{S}_b^{\text{cross}}$ of ursegments that intersect $h$, (ii) the set $\mathcal{S}_b^{\text{above}}$ of ursegments that pass above $h$, and (iii) the set $\mathcal{S}_b^{\text{below}}$ of ursegments that pass below $h$. Note that some of these subsets may be empty. If $\mathcal{S}_b^{\text{cross}}$ is not empty for one of these bundles, then we add the arc $(h_b, h)$ to the output graph $G$.

For each (non-empty) subset of type (ii) we create a new bundle $(\mathcal{S}_b^{\text{above}}, h_b)$, and similarly we create new bundles for the subsets $\mathcal{S}_b^{\text{below}}$.

Finally we construct one new bundle $b' = (S_{b'}, h)$, where we merge together all the subsets $\mathcal{S}_b^{\text{cross}}$. Note that we should delete ursegments from $b'$ that end at $h$, and that we need to add ursegments that start at $h$. We do these deletions and additions while sweeping through the pixel $h$—see the next step.

*Step 3: Update the status structures.*

We first create the trees $\mathcal{T}_b$ for the new bundles $b$, but still using the old ordering along $\ell$, that is, the ordering before $\ell$ crossed $h$. For the new bundles of type (ii) and (iii) this can be done in $O(\log n)$ time, by splitting the tree $\mathcal{T}_{\hat{b}}$ of the old bundle $\hat{b}$ that generated $b$. For the bundle that collects all ursegments crossing $h$, we have to merge several (pieces of) old trees into one, taking time $O(\log n)$ per old bundle crossing $h$.

Note that the ordering of the ursegments in bundles of type (ii) and (iii) along $\ell$ after $\ell$ crosses $h$ is the same as the ordering before along $\ell$ before $\ell$ crosses $h$, because all these ursegments miss $h$. Hence, we only need to update the tree $\mathcal{T}_{b'}$ of the new bundle $b'$ starting at $h$. At this stage we also add the ursegments starting at $h$. This can be done by sweeping the part of the arrangement of ursegments in $S^+$ inside $h$, using and updating the tree $\mathcal{T}_{b'}$ while we go. During this sweep, ursegments that end in $h$ will be deleted, and ursegment starting in $h$ will be inserted into $\mathcal{T}_{b'}$. Moreover, when we have swept over $h$ completely, the tree $\mathcal{T}_{b'}$ has the correct ordering. The time for this is $O((|S_h^+| + I_h) \log n)$, where $S_h^+$ is the set of ursegments of positive slope crossing $h$, including those starting at $h$, and $I_h$ is the number of intersections of $S_h^+$ inside $h$.

Finally, we update $\mathcal{T}$. We first delete all delimiting ursegments of the bundles that have been destroyed because they crossed $h$. Next we insert the delimiting ursegments of the new bundles, in $O(\log n)$ time per bundle. Since we have the updated trees $\mathcal{T}_b$ of the new bundles available at this point, we can find the delimiting ursegments of the new bundles in $O(\log n)$ time per bundle.

This finishes the description of the actions taken at each event point, and thereby the description of the algorithm. It remains to analyze the running time.

To compute the hot pixels $H$ we use standard plane sweep requiring $O((n + I) \log n)$ time where $I$ is the number of intersection points in the arrangement $\mathcal{A}(\mathcal{S})$. (This stage could be carried out more efficiently by more involved algorithms [1, 2], but the later stages require that amount of time.) Notice that the number of hot pixels is at most $O(n + I)$.

Each event point in the plane-sweep algorithm is a hot pixel, so it corresponds to a vertex in the output graph. Handling a hot pixel $h$ takes time $O(\log n)$ per bundle incident to $h$, which is equal to the degree of the corresponding vertex in the output graph, plus $O(\log n)$ time per intersection point inside $h$. Since the graph is planar, the total degree is linear in the number of vertices, which is $O(n + I)$. As the total number of intersections is $I$, the total time spent by the algorithm is $O((n + I) \log n)$. We get the following theorem.

**Theorem 2.1** *Given a collection $\mathcal{S}$ of $n$ line segments in the plane, the arrangement $\mathcal{A}(\mathcal{S})$ can be snap-rounded in time $O((n + I) \log n)$, where $I$ is the number of intersection points in $\mathcal{A}(\mathcal{S})$.*

# 3 Simplifying a rounded arrangement

As stated in the introduction and illustrated in Fig. 2, snap-rounded arrangements can contain degree-2 vertices that do not correspond to segment endpoints. In this section we show that we can simply delete such vertices—that is, replace the two incident edges with a single edge—without losing the nice properties of snap rounding.

Let $\mathcal{S}$ be a collection of segments in the plane, which, as before, we call ursegments. We assume we are given a grid onto which we want to round the arrangement $\mathcal{A}(\mathcal{S})$ induced by the ursegments. Let $P$ be some collection of pixels that includes the pixels containing the endpoints of the ursegments. We say that we *round* $\mathcal{A}(\mathcal{S})$ *to* $P$ if we replace every segment $s \in \mathcal{S}$ by the polyline that connects the centers of the pixels in $P$ that $s$ passes through; we denote the resulting arrangement by $\mathcal{A}_P(\mathcal{S})$.

Traditional snap rounding uses for $P$ the collection $H$ of hot pixels defined by $\mathcal{S}$, that is, the collection of pixels containing a vertex of $\mathcal{A}(\mathcal{S})$. We propose to use a smaller set for $P$. Define a hot pixel to be *red* if it either contains an endpoint of an ursegment, or if its degree in $\mathcal{A}_H(\mathcal{S})$ is at least three, and let $R$ be the collection of red pixels. We call $\mathcal{A}_R(\mathcal{S})$ the *simplified snap-rounded arrangement* of $S$. The remaining hot pixels—the ones that have degree two and do not contain an endpoint of an ursegment—are called orange; these are not used by our scheme.

Note that we can obtain $\mathcal{A}_R(\mathcal{S})$ from $\mathcal{A}_H(\mathcal{S})$ by replacing every polygonal chain $v_1, v_2, \ldots, v_k$ whose first and last vertex are red and whose inner vertices are orange by the single segment $v_1v_k$. Hence, we can compute the simplified rounded arrangement $\mathcal{A}_R(\mathcal{S})$ in $O((n+I)\log n)$ time, where $I$ is the number of intersection between the segments in $\mathcal{S}$: first compute $\mathcal{A}_H(S)$ with the algorithm of the previous section, and then discard the orange vertices. Next we show that this simplified rounded arrangement retains all the nice properties of traditional snap rounding.

**Theorem 3.1** *Let $\mathcal{S}$ be a set of segments in the plane. Then the simplified snap-rounded arrangement $\mathcal{A}_R(\mathcal{S})$ has the three properties of traditionally snap-rounded arrangements—fixed-precision representation, geometric similarity, and topological similarity—and, in addition, the non-redundancy property.*

**Proof:** Guibas and Marimont [6] showed that $\mathcal{A}_H(\mathcal{S})$ has the traditional properties. We will transform $\mathcal{A}_H(\mathcal{S})$ into $\mathcal{A}_R(\mathcal{S})$ by discarding the orange vertices one by one, and show that these properties are invariant under the transformation. After having discarded all orange vertices, we clearly have the non-redundancy property, and so the proof will be finished.

In each step of the transformation, we have a set $P$ of pixels with $H \subseteq P \subsetneq R$, and an orange pixel $p \in P$ that we want to discard. In other words, we want to transform $\mathcal{A}_P(\mathcal{S})$ into $\mathcal{A}_{P-\{p\}}(\mathcal{S})$. Let $q_1$ and $q_2$ be the two neighboring vertices of $p$ in $\mathcal{A}_P(\mathcal{S})$. Discarding $p$ means that we replace the polyline $q_1, p, q_2$ by the line segment $q_1q_2$.

To prove geometric similarity, we observe that any ursegment $s$ is still represented by a polyline connecting pixels that it passes through. Hence, we can use the same argument as Guibas and Marimont: any link in the polyline for $s$ has its two endpoints at pixel centers of pixels intersected by $s$, so the endpoints are contained in the Minkowski sum of $s$ and a pixel

centered at the origin. By convexity of Minkowski sums of a pair of convex objects, this means that the representation $s^*$ of $s$ is contained in the Minkowski sum as well.

It remains to establish fixed-precision representation and topological similarity. We will prove that we can continuously transform $q_1, p, q_2$ to $q_1q_2$ without passing over any vertex of $\mathcal{A}_P(\mathcal{S})$. Since $\mathcal{A}_P(\mathcal{S})$ has the topological similarity property, this means that $\mathcal{A}_{P-\{p\}}(\mathcal{S})$ also has this property. Moreover, if no segment passes over a vertex, no new intersections are created, and so we also still have the fixed-precision representation property. To prove that we can do the transformation, we observe that the triangle $\Delta$ defined by $q_1, p, q_2$ cannot contain a vertex of $\mathcal{A}_P(\mathcal{S})$ besides $q_1, p, q_2$ themselves. Indeed, let $s$ be any ursegment passing through $q_1$, $p$, and $q_2$ (there must be at least one such segment) and let $\hat{s}$ be the Minkowski sum of $s$ and a pixel centered at the origin. Then $q_1, p, q_2$ are all contained in $\hat{s}$, and so $\Delta \subset \hat{s}$ by convexity. Hence, $s$ will cross any pixel whose center lies in $\Delta$. Because $q_1$ and $q_2$ are neighbors of $p$ in $\mathcal{A}_P(S)$, this implies that no pixel in $P$ can have its center in $\Delta$. It follows that $\mathcal{A}_P(\mathcal{S})$ does not have a vertex in $\Delta$, so we can transform the polyline $q_1, p, q_2$ into $q_1q_2$ without passing over a vertex, as claimed. □

# 4 Concluding remarks

We have presented an intersection-sensitive algorithm to compute the snap-rounded arrangement of a given set of $n$ segments in the plane. Our algorithm runs in $O((n + I) \log n)$ time, where $I$ is the number of intersections between the segments in $S$. Thus considerably improving, in the worst case, over the best previously known solutions. We also showed that one can simplify the arrangement so that it does not have any degree-2 vertices except at shared endpoints of the rounded segments.

Our work suggests a number of directions for future research. First of all, it would be ideal to have an output-sensitive algorithm for snap rounding, that is, an algorithm whose complexity is sensitive to the complexity of the rounded arrangement. Neither our algorithm nor the algorithm by Guibas and Marimont [6] achieves this. Another interesting direction is to define a snap-rounding scheme with more good properties, while retaining the ones of our scheme. For example, it would be nice if one could guarantee a minimum separation between vertices and edges. The iterated snap-rounding of Halperin and Packer [8] achieves this, but their scheme is only defined algorithmically and it may result in very poor preservation of the geometric similarity. Another property that would be desirable is that the intersection of any two rounded segments is either a point or a connected subset of shared links; in the current schemes two rounded segments can meet and diverge many times.

# References

[1] I. J. Balaban. An optimal algorithm for finding segment intersections. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 211–219, 1995.

[2] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, 1992.

[3] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms (2nd edition).* MIT Press, 2001.

[5] D. H. Greene. Integer line segment intersection. Unpublished Manuscript.

[6] L. Guibas and D. Marimont. Rounding arrangements dynamically. *Internat. J. Comput. Geom. Appl.*, 8:157–176, 1998.

[7] D. Halperin and E. Leiserowitz. Controlled perturbation for arrangements of circles. *Comput. Geom. Theory Appl.*, 14:277–310, 2004.

[8] D. Halperin and E. Packer. Iterated snap rounding. *Comput. Geom. Theory Appl.*, 23:209–225, 2002.

[9] J. Hobby. Practical segment intersection with finite precision output. *Comput. Geom. Theory Appl.*, 13:199–214, 1999.

[10] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[11] V. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. In D. Kapur and J. L. Mundy, editors, *Geometric Reasoning.* North-Holland, Amsterdam, Netherlands, 1988.

[12] E. Packer. *Finite-Precision Approximation Techniques for Planar Arrangements of Line Segments.* Master's thesis, Dept. Comput. Sci., Tel-Aviv Univ., 2002.

[13] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[14] S. Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.

[15] K. Sugihara. On finite-precision representations of geometric objects. *J. Comput. Syst. Sci.*, 39:236–247, 1989.

[16] C. K. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.

[17] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry (2nd edition)*, chapter 41, pages 927–952. CRC Press LLC, Boca Raton, FL, 2004.