Computational Geometry 46 (2013) 403–416

Contents lists available at SciVerse ScienceDirect

Computational Geometry: Theory and Applications

www.elsevier.com/locate/comgeo

Stable snap rounding

John Hershberger

Mentor Graphics Corp., 8005 SW Boeckman Road, Wilsonville, OR 97070, United States

ARTICLE INFO

Article history:
Received 9 July 2011
Received in revised form 17 January 2012
Accepted 27 February 2012
Available online 7 November 2012
Communicated by F. Hurtado and M. van Kreveld

Keywords:
Snap rounding
Robust geometric computation

ABSTRACT

Snap rounding is a popular method for rounding the vertices of a planar arrangement of line segments to the integer grid. It has many advantages, including minimum perturbation of the segments, preservation of the arrangement topology, and ease of implementation. However, snap rounding has one significant weakness: it is not stable (i.e., not idempotent). That is, applying snap rounding to a snap-rounded arrangement of n segments may cause additional segment perturbation, and the number of iterations of snap rounding needed to reach stability may be as large as $\Theta(n^2)$.

This paper introduces *stable snap rounding*, a variant of snap rounding that has all of snap rounding's advantages and is also idempotent. In particular, stable snap rounding does not change any arrangement whose vertices are already grid points (such as those produced by stable snap rounding or standard snap rounding).

1. Introduction

Geometric algorithms are often described in terms of abstract lines, points, etc., whose positions and spatial relationships can be computed exactly. However, any implementation must represent these geometric objects using finite precision, characterized by the number of bits used. Geometric coordinates may be represented as floating point numbers, rational numbers, or, commonly, as fixed-precision integers.

The control flow of a geometric algorithm is determined by the relationships between geometric objects, with the relationships typically represented by the signs of algebraic functions of the object coordinates. The running time of an algorithm depends significantly on the algebraic degree of the functions it uses, because the number of bits needed to evaluate a function accurately may be as large as $\Theta(db)$, where d is the algebraic degree of the function and b is the number of bits in the arguments to the function.

If a geometric algorithm constructs new geometric objects based on its inputs, the new objects typically require more bits to represent accurately than the inputs. For example, the intersection of two line segments whose endpoints have b-bit integer coordinates has rational coordinates requiring a total of $5b + O(1)$ bits. It is clear that if algorithms are cascaded, i.e., if the outputs of one algorithm are fed as inputs to another, then the precision required for accuracy can quickly become unmanageable. A cascade of k line segment intersections, each using line segments whose endpoints are determined by the previous stage in the cascade, would require $5^k(b + O(1))$ bits to represent the final outputs accurately in terms of the original inputs. For this reason, geometric algorithms often round their outputs to the same precision as their inputs.

A rounded geometric result must be close to the true result, for some definition of "close", if it is to be useful. For example, the rounded result may be the correct result for a problem whose inputs are somehow close to the actual inputs—this is the *backward error analysis* of numerical analysis [22]. Alternatively, the rounded result may accurately represent the combinatorial structure of the true result, but with lower-precision coordinates.

E-mail address: john_hershberger@mentor.com.

0925-7721/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.comgeo.2012.02.011

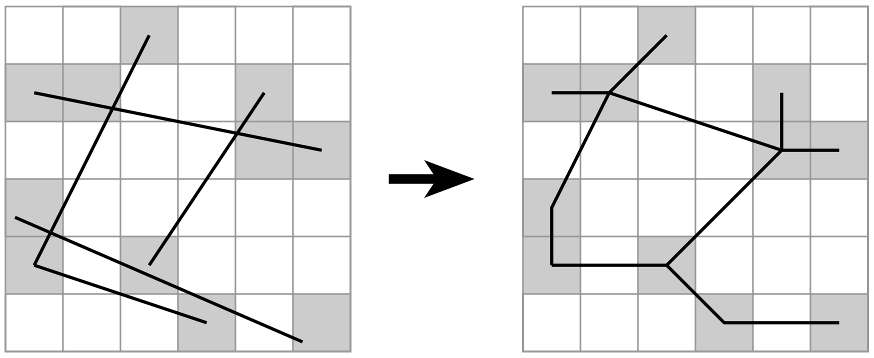

Fig. 1. Snap rounding deforms each original segment to pass through the centers of the hot pixels (shaded) that it intersects.

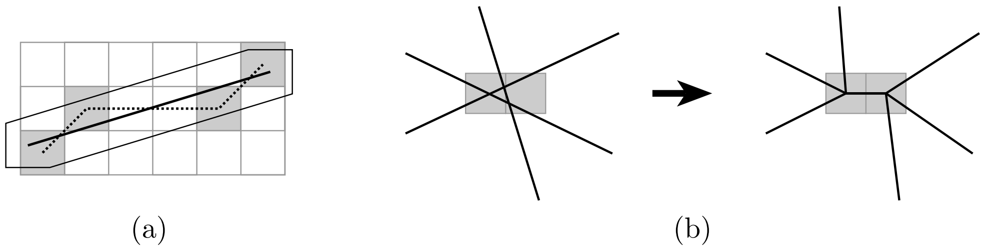

Fig. 2. (a) The snap rounded image of an ursegment s lies within half a pixel of s. (b) The topology of $\mathcal{A}(S)$ is preserved up to collapsing of features.

Greene and Yao [9] were the first computational geometers to propose an algorithm for rounding a line segment arrangement (the planar subdivision induced by a collection of line segments) to the integer grid. Their approach deforms each segment to a polygonal path, moving each of its intersections to the nearest grid point while constraining the resulting path not to pass over any other grid point. At the same time each moving intersection acts as a "hook" to pull other segments to the same grid point, ensuring that the topological relationships between segments and vertices are preserved. This algorithm guarantees accurate representation of the arrangement, but it may increase the total number of vertices in the arrangement by a logarithmic factor.

Milenkovic [19] presented an algorithm for rounding an arrangement of line segments as part of his "double precision geometry" framework. His approach replaces each line segment by a nearby line, then applies an algorithm for rounding the line arrangement. The algorithm maintains topological correctness for the line arrangement by replacing each line by a shortest path among the rounded positions of nearby arrangement vertices. This algorithm does not suffer the logarithmic blowup of Greene and Yao's approach, but it may not preserve the topology of the original input.

In both of these algorithms, the interaction between each arrangement vertex and each nearby segment is computed explicitly. *Snap rounding*, invented independently by Greene [8] and Hobby [15], simplifies rounding by separating the computation of arrangement vertices from segment rounding. To distinguish between the original segments and their snap rounded counterparts, the original input segments are called *ursegments* ("ur" is both an abbreviation for "UnRounded" and a German prefix meaning "original"). The definition of snap rounding is algorithm-independent:

1. A *pixel* is a unit square centered on an integer grid point.
2. A *hot pixel* is a pixel that contains an endpoint of an ursegment or the intersection of two noncollinear ursegments. (Collinear ursegments do not create internal hot pixels.)
3. Each ursegment s is *snap rounded* by replacing it by a polygonal path linking the centers of the hot pixels s intersects, as illustrated in Fig. 1.

Snap rounding makes two very important quality guarantees [11]. First, the rounded version of each ursegment s lies within half a pixel distance of s; that is, the rounded segment lies inside the Minkowski sum of s with the unit pixel. (See Fig. 2(a).) Second, if $\mathcal{A}(S)$ is the arrangement of a set of ursegments S, then the topology of $\mathcal{A}(S)$ is preserved in the rounded arrangement, up to collapsing of features. A face of $\mathcal{A}(S)$ may collapse to a line segment or a point, and a line segment may collapse to a point, but no face will reverse orientation; likewise, every pair of intersecting ursegments in $\mathcal{A}(S)$ maps to a pair of rounded segments that intersect, and any vertical grid line that intersects two ursegments in some order does not intersect their rounded images in the opposite order. (See Fig. 2(b).)

Canonicity is another important snap rounding feature: the result of snap rounding depends only on the ursegments, and not on any algorithmic choices or order of operations. Canonicity is important because it simplifies the proof of metric and topological accuracy. By contrast, the *unprincipled* rounding scheme sometimes used in practice (detect segment

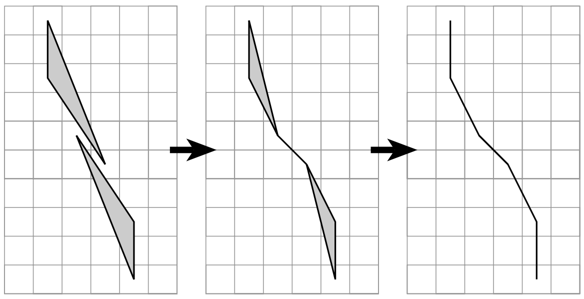

Fig. 3. Repeated snap rounding causes these triangles to collapse to line segments.

intersections in any order; repair each intersection by rounding it to the nearest grid point and break each of the two segments at the grid point; repeat) can cause the rounded representation to drift arbitrarily far from its ursegment, and may alter the topology of the arrangement. *Implementability* is a final advantage of snap rounding: many efficient algorithms are known to compute a snap rounded arrangement of line segments [2,7,8,11,14,15].

Although snap rounding has many advantages, it suffers from one important weakness: it is not *stable* (not idempotent). In other words, applying snap rounding to its own output may change the result. Almost equivalently, the modified segments that snap rounding produces may intersect hot pixels that the original segments did not. (The set of hot pixels does *not* change in repeated applications of snap rounding.) Although a single application of snap rounding guarantees that segments do not drift by more than half a pixel (contrasting with the unprincipled scheme), the problem of drift persists at a higher level: segments may drift arbitrarily far when snap rounding is applied repeatedly. See Fig. 3, in which repeated snap rounding causes two intersection-free triangles to collapse.

Higher-level features of an arrangement, such as polygonal faces, may be drastically changed by the rounding of their bounding segments. Even though “collapsing of features” is an expected consequence of snap rounding, instability of previously rounded arrangements sometimes proves surprising in practice. When a program depends on the number and connectivity of a set of polygons remaining stable under repeated rounding, “surprise” usually takes the form of a program crash.

One possible way to avoid snap rounding’s instability is to store the ursegments with the rounded arrangement and associate each rounded segment with the ursegment from which it derives. Subsequent rounding operations refer to the original ursegments, not their rounded images, eliminating drift. In practice, however, problems caused by drift are rare, and the overhead of storing both the ursegments and the rounded arrangement is unacceptably large.

Iterated Snap Rounding (ISR), introduced by Halperin and Packer [13], ensures that no rounded segment intersects a hot pixel. The goal of ISR is to avoid the near-degenerate output created when a snap-rounded segment passes near a snap-rounded vertex, i.e., through a hot pixel. (Under ordinary snap rounding, rounded segments and vertices may be arbitrarily close, if the coordinates are large enough.) As a side effect, ISR solves the problem of snap rounding instability. ISR is a very simple idea: apply snap rounding to an arrangement repeatedly until no further perturbation occurs. This is canonical and preserves the topology of the input, but it is not accurate: the rounded version of an ursegment s may be arbitrarily far from s. A naïve implementation is potentially very slow; the more sophisticated implementation proposed in [13] is still both more complicated and asymptotically substantially slower than a standard sweepline implementation of ordinary snap rounding ($O(n\log n + |\mathcal{A}(S)| + (\sum_{h\in\mathcal{H}} |h|)^{2/3}|\mathcal{H}|^{2/3+\epsilon} + \sum_{h\in\mathcal{H}} |h|)$ vs. $O(|\mathcal{A}(S)|\log n + \sum_{h\in\mathcal{H}} |h|)$). The sophisticated algorithm also requires the entire arrangement to be represented in memory simultaneously, whereas a sweepline algorithm needs to represent explicitly only the part of the arrangement that intersects a single vertical line; the rest of the input and output can be streamed from and to secondary storage or a compressed representation. For the large arrangements that arise in applications such as electronic circuit design [3,17,23], the difference in memory usage can have a very large impact on runtime.

Iterated Snap Rounding with Bounded Drift (ISRBD), an algorithm due to Packer [21], addresses the inaccuracy of ISR. The idea of ISRBD is to analyze which ursegments will drift unacceptably far under ISR, then introduce additional hot pixels that will prevent those ursegments from moving more than a prescribed amount. This scheme has two disadvantages: first, it is even slower than the already slow ISR; second, its output is noncanonical, since the locations of the additional hot pixels are chosen by the algorithm to meet a prescribed error tolerance.

To address the instability of snap rounding without the disadvantages of ISR and ISRBD, this paper introduces a new rounding scheme called *Stable Snap Rounding*. This algorithm slightly modifies the definition of snap rounding: instead of deforming each ursegment to pass through the center of every hot pixel it intersects, stable snap rounding divides the hot pixels into two classes and uses different rules to control ursegment deformation inside the two classes. Section 2 gives the full details of the scheme. As a consequence of this simple rule change, stable snap rounding achieves the following advantages:

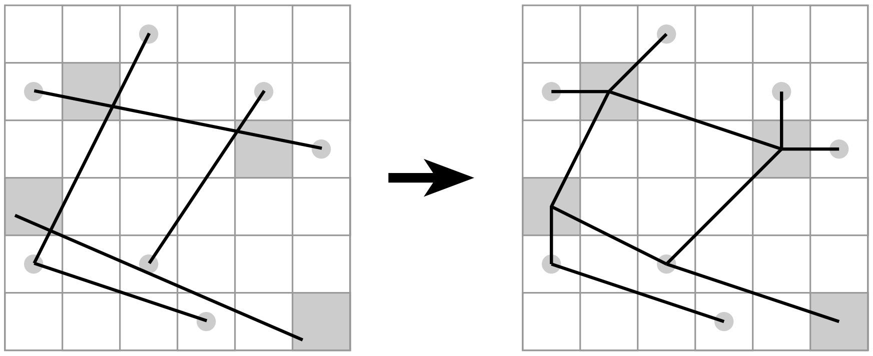

Fig. 4. Stable snap rounding applied to the example of Fig. 1. Magnet pixels are fully shaded, pins are shown as shaded dots. Note that the lower left segment intersects the pin pixel above it, but does not round to its center.

Idempotence: Applying stable snap rounding to the output of either stable snap rounding or ordinary snap rounding has no effect.[1]

Metric and topological accuracy: Stable snap rounding has the same guarantees of maximum segment drift and topology preservation as the original snap rounding scheme.

Canonicity: Stable snap rounding is defined using only the original segments, so its output is independent of any implementation choices.

Efficiency: Stable snap rounding can be implemented using a simple sweepline algorithm that is nearly as efficient as a sweepline algorithm for standard snap rounding [15]. More sophisticated input- and output-sensitive algorithms for standard snap rounding [2,7,14] can also be extended to work for stable snap rounding.

The rest of this paper presents stable snap rounding in detail. Section 2 defines the stable snap rounding scheme and proves its accuracy, canonicity, and idempotence. Section 3 addresses efficiency, showing that efficient algorithms for standard snap rounding can be extended to work for stable snap rounding with the same performance guarantees. Section 4 describes an implementation of stable snap rounding and presents experimental results.

2. Definitions and properties

Snap rounding is defined in terms of *pixels*, which are the unit squares centered at integer grid points. To ensure that each point of the plane belongs to exactly one pixel, pixels are defined to be closed at the left and bottom edges and open at the right and top edges. The bottom left corner is closed and the other three are open. An ursegment intersects a pixel iff it intersects its interior or a closed boundary point. In ordinary snap rounding, *hot pixels* are exactly those pixels that contain ursegment vertices or intersections of noncollinear ursegments. Stable snap rounding refines that definition, without creating any new hot pixels. It partitions the hot pixels of ordinary snap rounding into two classes:

Magnet pixels contain at least one vertex of the ursegment arrangement that is not an integer grid point.
Pin pixels contain ursegment vertices and intersections only at the pixel centers (i.e., at integer grid points).

The centers of the two pixel types (the associated integer grid points) are called *magnets* and *pins*. These names correspond to the rôles the pixels and their centers play in the following rounding scheme, illustrated in Fig. 4:

> **Stable Snap Rounding:** Each ursegment s is replaced by the polygonal path that passes through the centers of all magnet pixels that s intersects and connects consecutive magnets by the shortest path homotopically equivalent to s with respect to the pins.

The rounded image of an ursegment s is denoted by $\text{SSR}(s)$. For a given ursegment s, let $M(s)$ be the set of magnet pixels that s intersects, plus the pin pixels whose centers s touches (including at its endpoints, if s is defined by integral vertices). We will also use $M(s)$ to denote the centers of these pixels. Each ursegment passes above and below a certain set of pins; the rounded image of s is the shortest possible path among all paths that connect the pixel centers of $M(s)$ in order and have the same sets of pins above and below. Intuitively, one can think of s as a rubber band, pulled from its initial position to the centers of the pixels in $M(s)$ and constrained not to pass over pins. The rounded image $\text{SSR}(s)$ visits the points of $M(s)$ in the same order as s does.

[1] Unlike ISR, stable snap rounding may produce rounded segments that intersect hot pixels, passing arbitrarily close to the pixel centers, but the modified rounding rules ensure idempotence.

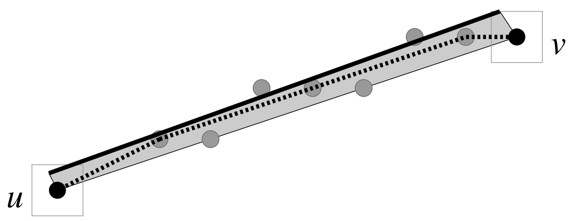

Fig. 5. The only pins that $\textsc{ssr}(\overline{s(u)s(v)})$ touches lie inside the shaded convex hull of $u, v, s(u)$, and $s(v)$.

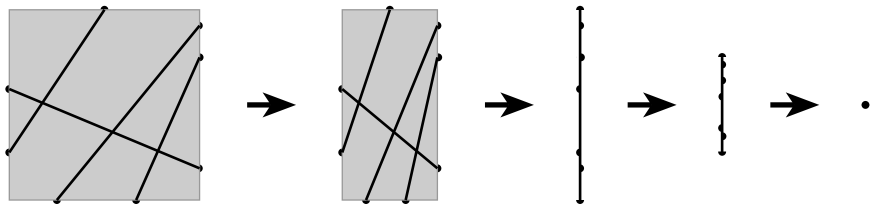

Fig. 6. Affine transformation of a magnet pixel does not affect topological relationships inside it.

Define the *unit pixel* U to be the pixel centered at the origin $(0, 0)$. The error introduced by snap rounding an ursegment s is bounded using the Minkowski difference (vector difference) of s and U.

Lemma 2.1. *The rounded image* $\textsc{ssr}(s)$ *of an ursegment* s *lies inside the Minkowski difference of* s *and the unit pixel, i.e.,* $s \ominus U$.

Proof. Consider the portion of $\textsc{ssr}(s)$ between consecutive pixel centers $u, v \in M(s)$. Define the *sausage* of s to be the Minkowski difference $s \ominus U$. See Fig. 2(a). Any pixel that s intersects has its center inside the sausage. Therefore, u and v are inside the sausage; because the sausage is convex, the segment $\overline{uv}$ also lies inside the sausage. (This observation finishes Guibas and Marimont's proof of the lemma corresponding to this one for ordinary snap rounding.) Let $s(u)$ and $s(v)$ denote the closest points on s to the grid points u and v under the L_∞ distance. Thus $u \in s(u) \ominus U$ and $v \in s(v) \ominus U$. The rounded image of $\overline{s(u)s(v)}$ can be obtained by a continuous deformation that moves $s(u)$ to u and $s(v)$ to v while maintaining the shortest path between them that is homotopically equivalent to $\overline{s(u)s(v)}$. The only pins that the final rounded path can touch between u and v lie inside the convex hull of u, v, $s(u)$, and $s(v)$. Since all four of these points lie inside the sausage, so does the convex hull, and hence so does $\textsc{ssr}(\overline{s(u)s(v)})$. See Fig. 5. □

Lemma 2.2. *The arrangement of snap-rounded segments is homotopic to the arrangement of ursegments, up to collapsing of features.*

Proof. The proof follows the approach of Guibas and Marimont [11], slightly modified as noted below. Each ursegment is partitioned into *edges* by adding *nodes* at vertices of the ursegment arrangement (endpoints and crossings) and at intersections of ursegments with magnet pixel boundaries. If an ursegment intersects two adjacent magnet pixels at a common boundary, it has two co-located nodes, one belonging to each pixel, with a zero-length edge joining them.

We define a continuous, simultaneous deformation of all the ursegments such that the homotopy type of the arrangement is preserved at all times, and such that the result of the deformation is the stable snap rounded arrangement. During the deformation, each edge of each ursegment is represented by a polygonal path of one or more segments. The homotopy type of the arrangement can change only if one of these segments passes over an endpoint of another such segment. We will show that this does not happen, and hence that the deformation preserves the homotopy type of the original arrangement, up to collapsing of features.

The deformation of an ursegment s occurs in two logically independent domains, one inside the magnet pixels and one outside. Each magnet pixel collapses to its center in two stages, first by a uniform x-contraction to the vertical bisector of the pixel, and second by a uniform y-contraction of that vertical segment to the pixel center. As a magnet pixel collapses, it deforms all the ursegment edges inside it uniformly by an affine transformation. Because during the collapse the arrangement of edges inside any magnet pixel is an affinely transformed image of the original arrangement inside the pixel, the homotopy type of those edges does not change—no segment passes over any node. See Fig. 6.

As magnet pixels collapse, the edges outside them (which connect nodes on different magnet pixel boundaries) deform. In the argument of Guibas and Marimont [11], each such edge is just the line segment connecting its two endpoint nodes. In this paper, the edge is actually a path, the shortest path between its endpoint nodes that is homotopic to the original edge with respect to the pins. That is, the edge deforms into a polygonal path that connects its node endpoints and has internal vertices at pins. See Fig. 7. Let us call the edges outside magnet pixels *external edges* and the nodes inside or on the

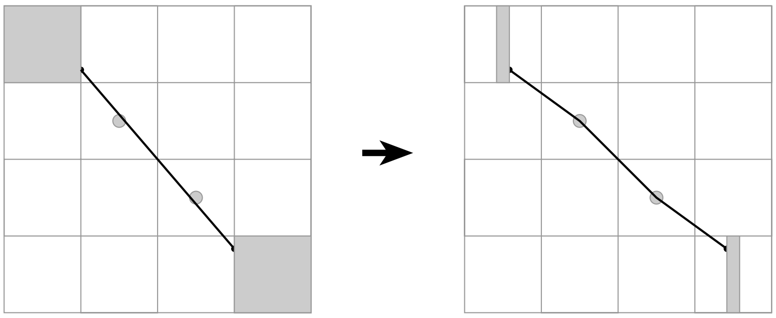

Fig. 7. An external edge deforms as the magnet pixels contract.

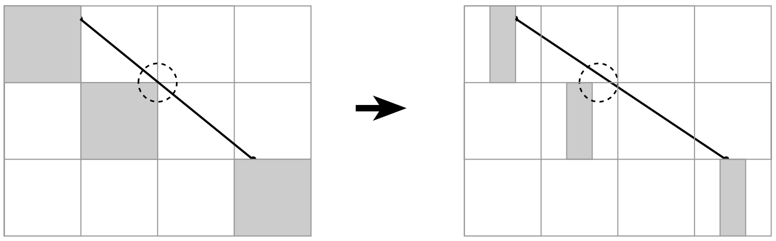

Fig. 8. A deforming external edge never contacts a contracting magnet pixel.

boundaries of magnet pixels *magnet nodes*. By construction, a deforming external edge does not cross pins, which are the only nodes not in the closure of magnet pixels. The endpoints of each segment of an external edge (the breakpoints of the polygonal path) are nodes; they are either pins or nodes on the boundary of magnet pixels.

To show that no external edge crosses any magnet node, we parameterize the contraction of the magnet pixels as in [11]: in the time interval $t \in [0, 1]$, magnet pixels contract uniformly to the vertical centerline of the pixel, and in the time interval $t \in [1, 2]$ the centerlines contract to the pixel centers. During $0 \leqslant t \leqslant 1$, nodes on the left and right boundaries of magnet pixels move horizontally with x-velocity $\pm\frac{1}{2}$. Nodes strictly inside their pixel's x-span move slower. Likewise, during $1 \leqslant t \leqslant 2$, nodes at the top and bottom of magnet pixels move with y-velocity $\pm\frac{1}{2}$; all other nodes move slower. During $0 \leqslant t \leqslant 1$, each point in the interior of a segment, which can be expressed as a fixed convex combination of the segment endpoints, moves left or right with x-speed (absolute value of x-velocity) at most $\frac{1}{2}$. (The y-coordinate of the point is fixed.) Since the edge is initially disjoint from every magnet pixel, it cannot cross any node inside a magnet pixel without first crossing over a corner of the containing magnet pixel. But the corner moves away from the segment with x-speed $\frac{1}{2}$, and so the segment can never catch up to it. See Fig. 8. Likewise, during $1 \leqslant t \leqslant 2$, the top and bottom ends of magnet pixels move inward with y-speed $\frac{1}{2}$, segment internal points move vertically with y-speed at most $\frac{1}{2}$, and so segments cannot cross over nodes inside magnet pixels. This establishes the lemma. □

The following theorem summarizes the features of stable snap rounding.

Theorem 2.3. *Stable snap rounding is accurate, canonical, and idempotent.*

Proof. Lemmas 2.1 and 2.2 establish the metric and topological accuracy of stable snap rounding. Because stable snap rounding is defined only by the features of the ursegment arrangement $\mathcal{A}(S)$, it is canonical. The output of stable snap rounding on the ursegment arrangement, denoted $\text{SSR}(\mathcal{A}(S))$, has all vertices at integer grid points. Therefore, if $\text{SSR}(\mathcal{A}(S))$ is fed as input to stable snap rounding, all hot pixels are pins—there are no magnets—and hence no segment is deformed. That is, $\text{SSR}(\text{SSR}(\mathcal{A}(S))) = \text{SSR}(\mathcal{A}(S))$. □

3. Algorithms

This section shows that many algorithms for ordinary snap rounding can be adapted to perform stable snap rounding. In particular, the basic sweepline algorithm of Hobby [15], the dynamical algorithm of Guibas and Marimont [11], and the output-sensitive algorithms of Goodrich et al. [7], de Berg, Halperin, and Overmars [2], and Hershberger [14] can all be adapted. The algorithms can be divided into three classes: single-segment algorithms, bundling algorithms, and dynamical algorithms. Each class is addressed in one of the subsections below.

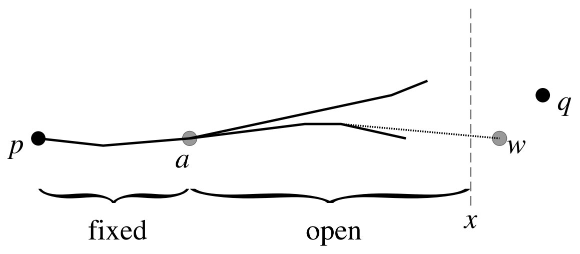

Fig. 9. The funnel algorithm for finding the shortest path from p to q. In this example, a and the pins before it (in the fixed part of the path) lie above the ursegment s. The addition of the first pin below s, just to the right of a, caused the funnel apex to move from p to a and fixed the path from p to a.

3.1. Single-segment algorithms

Hobby's implementation of snap rounding [15] is built on top of the classical Bentley–Ottmann sweepline algorithm for finding all intersections in an arrangement of line segments [1]. The Bentley–Ottmann algorithm finds the vertices of the arrangement of ursegments in left-to-right order; these vertices define the hot pixels.

Hobby's algorithm augments the Bentley–Ottmann sweepline by maintaining a second list of ursegments that shadows the main sweepline data structure and trails behind it by up to one grid unit. The overall algorithm advances the pair of sweeplines from left to right in phases. Each phase detects and processes all the hot pixels centered at a single integral x-coordinate $x = i$. At the start of the phase, the main sweepline and the shadow are synchronized and hold the segments that cross $x = i - \frac{1}{2}$ in their y-order at that x-coordinate. The first part of the phase advances the Bentley–Ottmann sweepline from $x = i - \frac{1}{2}$ to $x = i + \frac{1}{2}$, noting the events (ursegment vertices and crossings) and recording the hot pixels they induce. The second part of the phase finds the intersections of all ursegments with hot pixels centered at $x = i$. This takes $O(1)$ time per intersection, using a linear scan up and down in the shadow list from the location of each event. The algorithm computes the hot pixels intersected by each ursegment in left-to-right order. This allows it to generate the rounded edges of each ursegment on the fly, remembering the last visited hot pixel for each ursegment and writing out new edges as new hot pixel intersections are detected. The algorithm computes the snap rounded representation of all ursegments in time $O(|\mathcal{A}(S)| \log n + \sum_{h \in \mathcal{H}} |h|)$, where $\mathcal{A}(S)$ is the arrangement of the ursegments, $\mathcal{H}$ is the collection of all hot pixels, and $|h|$ is the number of ursegments intersecting a hot pixel $h \in \mathcal{H}$.

It is straightforward to adapt Hobby's algorithm to support stable snap rounding. Magnets are easy to distinguish from pins; only magnet pixels contain off-grid vertices of $\mathcal{A}(S)$. For each ursegment s, the stable snap rounding implementation stores in a list the pin pixels visited by s since the most recent element of $M(s)$. The pin pixels are stored in the left-to-right order visited. When the sweepline reaches the next element of $M(s)$, the implementation generates the rounded representation of the part of the ursegment whose pin pixels are stored in the list.

Generating the rounded representation for an ursegment s between two consecutive elements of $M(s)$ is a simple adaptation of an algorithm for computing a shortest path in a simple polygon [10,16]. If p and q are two consecutive elements of $M(s)$, the algorithm steps through the pin pixels between p and q from left to right, maintaining a *funnel* that represents all shortest paths with the desired homotopy type from p to the current x-coordinate. See Fig. 9. The shortest path from p to the apex a of the funnel is fixed, and the shortest paths from a to the current x-position are constrained to lie between two convex polygonal chains attached to the apex. When the x-position advances to a new pin w, the algorithm finds the tangent from w to the convex chains and inserts w into the appropriate chain, upper or lower, depending on whether s passes below or above w. The tangents are found by a linear scan over the convex chains, starting from the rightmost vertex on the same side of s (above/below) as w. Every pin scanned (except the last) is deleted from the chains. When q is reached, the process ends by finding the tangent from q to the funnel, determining the final shortest path from p to q.

When a new pin w is processed, some pins may be deleted from the convex chains and the apex a may move. The amount of work done in tangent-finding and pin-deleting is proportional to the number of pins deleted from the funnel, and so the algorithm is linear in the number of hot pixels s intersects. Pins that are dropped from the funnel without becoming part of the shortest path from a to p are grid points that would belong to a standard snap rounded representation of s but do not belong to the stable snap rounded representation.

Theorem 3.1. *Stable snap rounding can be computed for the arrangement $\mathcal{A}(S)$ of a set of n line segments S in time $O(|\mathcal{A}(S)| \log n + \sum_{h \in \mathcal{H}} |h|)$ by a simple extension of Hobby's sweepline algorithm [15].*

Other snap rounding algorithms that explicitly detect hot pixel/ursegment intersections in left-to-right order can be adapted to perform stable snap rounding using the approach described above. This includes the output-sensitive algorithm of Goodrich et al. [7], which runs in $O(\sum_{h \in \mathcal{H}} |h| \log n)$ time. The algorithm erases the parts of ursegments inside hot pixels, thereby eliminating the cost of detecting multiple intersections inside any given hot pixel. This gets rid of the dependence on $|\mathcal{A}(S)|$, the complexity of the input arrangement, which appears in the running time of Hobby's algorithm.

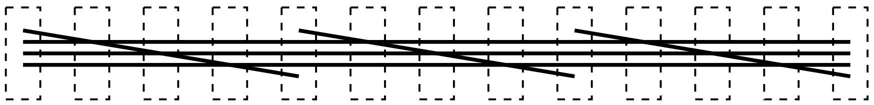

Fig. 10. The algorithms of [7] and [15] run in $\Theta(n^3 \log n)$ and $\Theta(n^3)$ time on this arrangement, because $\sum_{h\in\mathcal{H}} |h| = \Theta(n^3)$, even though the arrangement and the result of snap rounding it both have size only $O(n^2)$. (The arrangement is stretched vertically for clarity.)

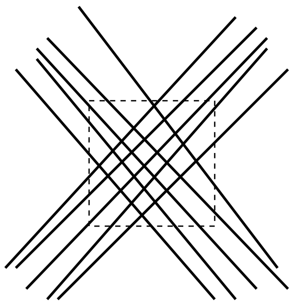

Fig. 11. If k segments form a crossbar inside h, then $is(h) = |h| = k$, the number of crossings is $\Theta(k^2)$, and $ed(h) = O(1)$.

To adapt the output-sensitive scheme to stable snap rounding, the algorithm is modified to erase ursegments only inside magnet pixels. This ensures that magnets and pins are properly identified and, if implemented carefully, still preserves the algorithm's running time. (One must be careful to spend $O(h \log n)$ time, not $O(h^2)$, in hot pixels where h ursegments pass through the center.) Just as in the adaptation of Hobby's algorithm, each active ursegment stores the sequence of pin pixels it intersects between its most-recently visited magnet and the current position of the sweepline. When a new intersection between a magnet pixel and an ursegment is detected, the algorithm finds the rounded subsegment induced by the stored sequence of pins in linear time.

Theorem 3.2. *Stable snap rounding can be computed for an arrangement of n line segments in time $O(\sum_{h\in\mathcal{H}} |h| \log n)$ by an extension of the algorithm of Goodrich et al. [7].*

3.2. Bundling algorithms

As de Berg, Halperin, and Overmars observed [2], algorithms whose running times include a term of the form $\sum_{h\in\mathcal{H}} |h|$ can be dramatically suboptimal. The input and output of snap rounding both have complexities at most $O(n^2)$: an arrangement of n segments has $O(n^2)$ intersections, and a snap rounded arrangement is an embedded planar graph on $O(n^2)$ hot pixel vertices. However, $\sum_{h\in\mathcal{H}} |h|$ may be as large as $\Theta(n^3)$, as illustrated in Fig. 10 (reproduced from [14]). To avoid the problem, de Berg, Halperin, and Overmars proposed an algorithm with running time $O(|\mathcal{A}(S)| \log n)$ that groups ursegments into *bundles* [2]. A bundle is a collection of ursegments that behave like a single segment between two consecutive hot pixels u and v: each segment in the bundle intersects u and v, does not intersect any hot pixel between u and v, and consequently does not intersect any other bundle segment between u and v.

Extending this bundling approach, Hershberger introduced two algorithms that compute a bundled representation of a snap-rounded arrangement without necessarily processing all the vertices of $\mathcal{A}(S)$. The running times of these algorithms are expressed in terms of quantities $is(h)$ and $ed(h)$, defined for all hot pixels $h \in \mathcal{H}$. The quantity $is(h)$ is the number of ursegments that have an intersection or an endpoint inside a hot pixel h. Note that $is(h) \leqslant |h|$, and the number of vertices of $\mathcal{A}(S)$ inside h may be as large as $\Theta(|h|^2)$. The quantity $ed(h)$ is the *edit distance* of the ursegments crossing h. Intuitively, it corresponds to the description complexity of the bundles crossing the boundary of h. It counts the number of ursegments with one endpoint inside h or with different adjacent ursegments on entry and exit from h. For example, if the ursegments crossing h are divided into two equal-size sets such that inside h each segment intersects every segment in the other set and no segment in its own, then the description complexity of the crossing pattern, $ed(h)$, is $O(1)$, even though $is(h) = |h|$ and the number of crossings is $\Theta(|h|^2)$. See Fig. 11. The edit distance $ed(h)$ is always $O(is(h))$, and sometimes much smaller. The two algorithms of [14] run in times $O(\sum_{h\in\mathcal{H}} is(h) \log n)$ and $O(\sum_{h\in\mathcal{H}} ed(h) \log n)$, respectively. Both algorithms produce a bundled representation of the snap rounded arrangement equivalent to that of [2].

To adapt bundling algorithms to use stable snap rounding, it is necessary to extend the framework of shortest paths among magnets and pins to handle bundles and not just individual ursegments.

The bundling algorithms of [2] and [14] compute the hot pixels and the snap rounded arrangement they induce in two separate phases. Stable snap rounding maintains this structure. The first phase computes the hot pixels as in the original algorithms [2,14], with the additional wrinkle that the hot pixels are further classified as either magnets or pins. (Since the algorithms of [14] do not detect all ursegment intersections explicitly, this is not entirely trivial. The key is to modify the

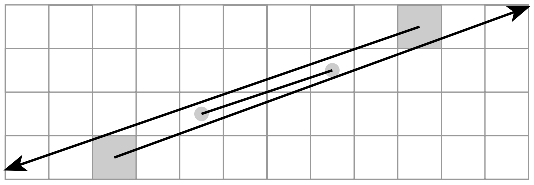
Fig. 12. The two long segments both intersect the shaded magnet pixels, but their bundles are distinct, because they pass on opposite sides of two pin pixels.

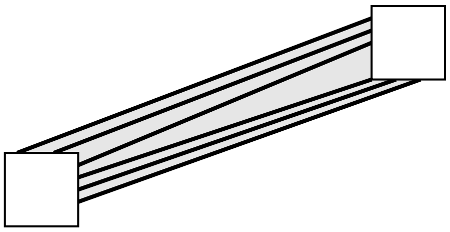
Fig. 13. A bundle region.

algorithms slightly so that if an ursegment s has an intersection at a hot pixel center, that does not prevent the detection of other intersections on s inside the same pixel.)

The second phase of the bundling algorithms threads the ursegments among the hot pixels using a plane sweep, grouping ursegments into bundles that connect pairs of hot pixels. Stable snap rounding modifies this by treating the two kinds of hot pixels differently during the threading process: magnet pixels are treated as full-size, but pin pixels are treated as infinitesimal in size (consisting only of the pins themselves). That is, a bundle is regarded as touching a pin pixel only if it hits the pin (the pixel center) itself.

The bundling algorithms sweep over the ursegments from left to right while maintaining a vertically ordered sequence of *one-ended bundles*—maximal contiguous groups of intersection-free ursegments, each group emanating from a common hot pixel at its left. At each column of hot pixels, the sequence of one-ended bundles is updated according to their interactions with the hot pixels. Segments that hit a hot pixel are split off from their bundles and finalized (recorded as part of a bundle connecting two hot pixels); segments that do not hit a hot pixel continue as part of a bundle, with the proviso that bundles may be split to ensure that the segments of a bundle are homotopically equivalent (a bundle may not straddle a hot pixel); segments that emerge from and extend to the right of a hot pixel initiate a new one-ended bundle. The only property that the threading algorithms of [2] and [14] need is the fact that ursegments intersect only inside the hot pixels. Therefore, the same algorithms work when pin pixels are regarded as infinitesimal.

The output of the second phase is a collection of ursegment bundles. Each bundle is defined by two pixels and a homotopy type. A bundle $B(u,v)$ consists of ursegments such that u and v are consecutive elements of $M(s)$ for each $s \in B(u,v)$, and all the bundle subsegments between u and v are homotopically equivalent with respect to the hot pixels. (It is possible that multiple bundles link a single pair of pixels u and v, distinguished by their homotopy types with respect to the pins. See Fig. 12 for a simple example. This is not possible in ordinary snap rounding, in which pin pixels are not treated as points. However, for convenience we use the notation $B(u,v)$ to refer to any such bundle, with the homotopy type specified by context.) A bundle $B(u,v)$ contains only the subsegments of its ursegments linking the pixels of u and v. Each bundle is represented as an ordered list of segments that supports insertion, deletion, search, concatenation, and splitting. The data structure uses partial persistence [5] to share substructures among these lists, so the total size of the data structure is asymptotically smaller than the sum of the list sizes.

Lemma 3.3. *Suppose that s and s' are segments of one bundle, t is a segment of another, and all three segments are intersected by a common vertical line ℓ. Then the intersections of ℓ with s and s' lie on the same side of t.*

Proof. The lemma holds by construction. The vertical line ℓ is equivalent to the sweepline at some time during the construction of the bundles. Contiguity of the one-ended bundles along the sweepline means that s and s' both lie consistently either above or below t.

For each bundle $B(u,v)$, one can bound the position of all its constituent segments by forming the polygon bounded by the extreme segments of the bundle (top and bottom, or left and right) and clipped to the exterior of the hot pixels. See Fig. 13. These bundle regions are interior-disjoint, and any two have a consistent vertical order, by Lemma 3.3. The vertical visibility relationships between bundles that are implicit in the sweepline order of bundles give a linear-size partial order on the bundle regions, which can be extended to a total order in linear time by topological sort [4]. To see this, observe that

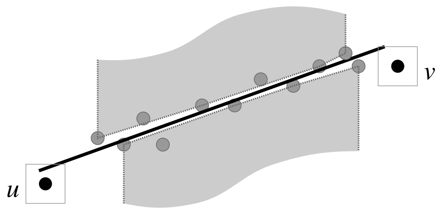

Fig. 14. $\textsc{ssr}(B)$ is sandwiched between the two convex hulls $H^-(B)$ and $H^+(B)$.

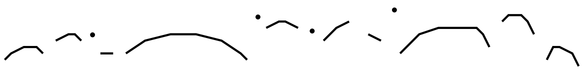

Fig. 15. The list L of convex chains.

each bundle region can be represented by a maximum-x-length segment inside it, and the result of Guibas and Yao [12] for vertical ordering of convex regions applies to these segments. This topological order, denoted $\prec$, has the property that for any two bundles B and B' with $B \prec B'$ and any two segments $s \in B$ and $s' \in B'$, no point of s' is vertically below any point of s.

Let $B = B(u, v)$ be a bundle. Pixels u and v are consecutive elements of $M(s)$ for every $s \in B$. Let $P^-(B)$ be the pins directly below B (below and in the x-span of $\overline{uv}$) and $P^+(B)$ be the pins directly above B. All the segments in B have the same stable snap rounded image between u and v, namely the shortest path from u to v that passes above or through all elements of $P^-(B)$ and below or through all elements of $P^+(B)$. Denote this path by $\textsc{ssr}(B)$. Since $P^-(B)$ and $P^+(B)$ are separated by every line segment in B, the only pins that can appear on $\textsc{ssr}(B)$ belong to the upper convex hull of $P^-(B)$ and the lower convex hull of $P^+(B)$, denoted by $H^-(B)$ and $H^+(B)$, respectively. See Fig. 14.

Lemma 3.4. *Let B and B' be bundles with $B \prec B'$, and let $p \in P^-(B)$. Then p appears on $\textsc{ssr}(B')$ only if $p \in H^-(B)$.*

Proof. Stable snap rounding preserves the order of segment intersections with a vertical grid line. (This follows from Lemma 2.2, as in [11].) Thus $\textsc{ssr}(B)$ lies between $\textsc{ssr}(B')$ and any point $p \in P^-(B)$ that also lies below B'. Such a point p can appear on $\textsc{ssr}(B')$ only if it also appears on $\textsc{ssr}(B)$, and this is possible only if $p \in H^-(B)$. $\square$

We would like to compute $\textsc{ssr}(B)$ for every bundle B. As Lemma 3.4 implicitly suggests, if $B \prec B'$ are two bundles intersected by a common vertical line, the subpath $\textsc{ssr}(B) \cap \textsc{ssr}(B')$ may be large. We want to compute $\textsc{ssr}(B)$ for all bundles in total time $O(|\mathcal{H}| \log n)$, so common substructures like $\textsc{ssr}(B) \cap \textsc{ssr}(B')$ must be computed efficiently and shared between the data structure representations of $\textsc{ssr}(B)$ and $\textsc{ssr}(B')$. This turns out to be easier to achieve with a bottom-to-top sweep over the bundles in $\prec$ order than with a left-to-right sweep in x-order.

Given the total order $\prec$ on the bundles, the stable snap rounding algorithm finds the convex hull of a subset of the pins above and below each bundle B. The subset is selected according to Lemma 3.4 such that no point that is omitted can possibly belong to $\textsc{ssr}(B)$.

The convex hulls that the algorithm computes below and above a bundle B are denoted $h^-(B)$ and $h^+(B)$. Lemma 3.5 (below) shows that although $h^-(B) \subseteq H^-(B)$ and $h^+(B) \subseteq H^+(B)$, every internal vertex of $\textsc{ssr}(B)$ belongs to either $h^-(B)$ or $h^+(B)$. The algorithm computes $h^-(B)$ for each B by a bottom-to-top scan through the bundles in $\prec$ order. The algorithm maintains an x-sorted list L of convex chains, initially empty, that contains fragments of $h^-(B')$ for all bundles $B' \prec B$. See Fig. 15. The list and the convex chains within it support the standard insert, delete, split, join, and search functions in logarithmic time [4]. Arc length computation is also needed later, and is easy to support within the same time bound. Here is the algorithm for computing the convex chains $h^-(B)$ for all bundles B:

Algorithm Bundle Hull Construction:

Input: A set of bundles, ordered by the $\prec$ relation; a set of pins, partitioned into subsets according to the bundles immediately above them—see below.

Output: A convex chain $h^-(B)$ for each bundle B such that $\textsc{ssr}(B)$ is the shortest path joining the bundle endpoints that lies above $h^-(B)$ and below the symmetrically defined convex chain $h^+(B)$.

Let L be an x-ordered list of convex chains, initially empty.
For each bundle B from bottom to top in $\prec$ order:
- Locate the x-coordinates of B's endpoints in L.
- Split (at most two) convex chains in L at the x-coordinates of B's endpoints.
- Let P be the set of pin points below B and not below any predecessor of B in $\prec$ order: $P = P^-(B) \setminus (\bigcup_{B' \prec B} P^-(B'))$.

- For every $p \in P$
  - Locate the x-coordinate of p in L.
  - Split the convex chain spanning p's x-coordinate, if any, at that position.
  - (*) Insert p into L as a singleton convex chain.
- If the number of convex chains in the x-interval of B is greater than one, merge them into a single upper convex hull. This convex hull is $h^-(B)$.
  (**) Points in the x-interval of B but not on $h^-(B)$ are dropped from L.

The chains $h^+(B)$ are computed symmetrically.

Lemma 3.5. *If $p \in P^-(B)$ belongs to* $\textsc{ssr}(B)$*, then $p \in h^-(B)$.*

Proof. Consider what happens to p during the process of computing all the $h^-()$ convex chains. Because p is directly below B, and all the bundles that overlap the x-interval of B below B are processed before B, p will be inserted into L at or before the processing of B (cf. Step (*)). Point p can be deleted from L only in Step (**). If it is deleted during the processing of B, it cannot belong to $H^-(B)$, because $h^-(B)$ is the hull of a subset of $P^-(B)$—if p does not lie on $h^-(B)$, *a fortiori* it cannot lie on $H^-(B)$. In this case $p \notin \textsc{ssr}(B)$, because only points on $H^-(B)$ and $H^+(B)$ can belong to $\textsc{ssr}(B)$. On the other hand, if p is deleted during the processing of B', for $B' \prec B$, then by Lemma 3.4, $p \notin \textsc{ssr}(B)$. □

The following lemma establishes the running time of the chain-construction algorithm:

Lemma 3.6. *The algorithm to construct the chains $h^-(B)$ for all bundles B can be implemented to run in time $O(|\mathcal{H}| \log n)$.*

Proof. The total number of bundles is $O(|\mathcal{H}|)$, because the bundles correspond one-to-one with the edges of an embedded planar graph whose vertices are the hot pixels. There are at most $|\mathcal{H}|$ pins, since pins are also hot pixel centers. The bundles directly above and below each pin can be determined in a plane sweep over the bundles and pins in $O(|\mathcal{H}| \log n)$ time. This determines the sets $P^-(B) \setminus (\bigcup_{B' \prec B} P^-(B'))$. Given these sets and the bundle order $\prec$, it is easy to argue that the Bundle Hull Construction algorithm has the stated complexity: Locating bundle endpoints and splitting L at their x-coordinates takes $O(\log n)$ time per bundle. Splitting L and inserting each new pin takes $O(\log n)$ time per pin. Finally, the construction of $h^-(B)$ by merging convex hulls takes $O(k \log n)$ time by standard techniques, where k is the number of hulls merged. Since hulls are created only by point insertion and by splitting with bundle endpoints and inserted points, and the total number of merges cannot exceed the total number of hulls created, the merge cost is at most $O(\log n)$ times the total number of bundles and pins. Putting it all together proves the lemma. □

Since the Bundle Hull Construction algorithm builds $h^-(B)$ for the bundles in $\prec$ order, and the construction of $h^-(B)$ may disassemble $h^-(B')$ for $B' \prec B$, the algorithm records each $h^-(B)$ using a persistent data structure [5]. It records $h^+(B)$ in the same way. Finally, the algorithm computes $\textsc{ssr}(B)$ from $h^-(B)$ and $h^+(B)$ for each bundle B using $O(1)$ logarithmic-time tangent computations. In more detail, for each bundle $B = B(u, v)$, the algorithm computes the tangents from u and v to $h^-(B)$ and $h^+(B)$, the inner common tangents between $h^-(B)$ and $h^+(B)$, and the segment $\overline{uv}$. Any of these segments that passes below $h^-(B)$ or above $h^+(B)$ is discarded. Then $\textsc{ssr}(B)$ is the shortest path from u to v through the constant-size geometric graph determined by the remaining segments, their endpoints, and the induced subarcs of $h^-(B)$ and $h^+(B)$, with each edge or arc weighted with its length. The representation of each $\textsc{ssr}(B)$ is stored in a persistent data structure. (This is similar to the way that bundle contents are represented as ordered lists of ursegments that are split and joined repeatedly as a sweepline passes over the bundles [2,14]. The segments in a bundle are collected into a single list only while the sweepline intersects the bundle, but the list can be represented for later inspection/use in a persistent data structure.)

Theorem 3.7. *Any algorithm that computes a bundled representation of ordinary snap rounding can be extended to perform stable snap rounding in $O(|\mathcal{H}| \log n)$ additional time.*

Proof. Lemma 3.6 bounds the construction time of $h^-(B)$ and $h^+(B)$ for all bundles by $O(|\mathcal{H}| \log n)$. Computing $\textsc{ssr}(B)$ requires $O(1)$ tangent computations on $h^-(B)$ and $h^+(B)$, at an additional cost of $O(\log n)$ time per bundle. □

We speculate that it may also be possible to compute $\textsc{ssr}(\mathcal{A}(S))$ during the same sweep that computes the bundles, instead of in a separate phase, as described here. This could reduce the algorithm's working storage and might produce a simpler implementation.

3.3. Dynamical algorithm

The dynamical snap rounding algorithm of Guibas and Marimont [11] combines a randomized arrangement data structure due to Mulmuley [20] with traditional snap rounding [8,15]. It maintains an explicit representation of the hot pixels and the bundles of ursegments connecting them. It uses multiple levels of hierarchy, with links between them, to speed point location in the arrangement. The running time of updates (ursegment insertions and deletions) depends on the number of hot pixels that the ursegment passes through or near, where "near" means within a half-pixel distance, i.e., within L_∞ distance 1/2, plus the number of ursegments passing through or near hot pixels created or destroyed by the ursegment insertion/deletion.

Instead of modifying the data structure of Guibas and Marimont to maintain stable snap rounding explicitly, we layer stable snap rounding on top of the data structure implicitly. That is, we augment the data structure of Guibas and Marimont to identify hot pixels as pins or magnets. This is easy to do by maintaining for each hot pixel h the number of ursegment crossings and endpoints that fall inside h but not at its center. Call this the *magnet count* of pixel h. The scheme of Guibas and Marimont detects all ursegment crossings explicitly, so it is easy to maintain the magnet counts of all the hot pixels. Any hot pixel with a nonzero magnet count is a magnet pixel; all other hot pixels are pin pixels.

Because stable snap rounding builds on the same hot pixels and ursegment/hot pixel incidences as traditional snap rounding, the addition of magnet counts to the Guibas–Marimont structure suffices to represent stable snap rounding implicitly. If the explicit representation of $\textsc{ssr}(s)$ is needed for any ursegment s, it can be computed in time linear in the number of hot pixels s intersects using the algorithm of Section 3.1. In fact, the portion of $\textsc{ssr}(s)$ between any two consecutive pixels $u, v \in M(s)$ can be computed in time proportional to the number of pin pixels s intersects between u and v. The entire stable snap rounded arrangement can be computed explicitly in $O(\sum_{h\in\mathcal{H}} |h|)$ time by this algorithm, based on the dynamical Guibas–Marimont structure, or in bundled form in $O(|\mathcal{H}| \log n + \sum_{h\in\mathcal{H}} |h|)$ time, including time to convert the dynamical structure to bundled form (cf. Section 3.2).

Theorem 3.8. *The dynamical snap rounding data structure of Guibas and Marimont [11] can be extended to maintain the stable snap rounded version of an arrangement implicitly with the same asymptotic complexity as the original data structure.*

If it is necessary to perform point location on the implicit stable snap rounded arrangement, first locate the query point q in either the arrangement of ursegments or its snap rounded version using the hierarchical data structure, then explicitly compute the relevant portions of $\textsc{ssr}(s)$ for ursegments s within half a pixel of q, i.e., within L_∞ distance 1/2. Locating q relative to $\textsc{ssr}(s)$ for each such s finishes the computation.

4. Implementation

Stable snap rounding has been implemented as part of a sweepline library used inside Mentor Graphics' Calibre family of electronic design automation products [18]. The implementation builds on a previous implementation of ordinary snap rounding similar in spirit to Hobby's (Section 3.1). The ursegment endpoints are integer grid points, so only ursegment intersections create magnet pixels. The input segments are stored in a proprietary compressed data structure ordered by their left endpoints. The output of snap rounding is written directly to a compressed data structure, and so only the active segments (those intersecting the current x-position of the vertical sweepline) need to be represented in uncompressed form. The implementation maintains the active ursegments in a searchable list ordered according to their intersections with the current position of the sweepline.

Sweepline events (deletions, insertions, crossings) are processed in phases, grouped according to their nearest integer x-coordinate. Thus the implementation processes the hot pixels belonging to a single vertical column in a single phase. Ursegment crossings are processed in a "topological sweep" fashion [6] in each phase, instead of in strict geometric order; this reduces the algebraic degree of the comparison predicates needed to sort the intersections. Each phase starts with the active segments in the sweepline properly ordered for some sweepline position $x = i - \frac{1}{2}$, for i an integer, and ends with the sweepline position advanced to $x = i + \frac{1}{2}$.

Each phase detects all the hot pixels in a single column, then finds all the intersections of ursegments with those hot pixels. This is done in time linear in the number of ursegment/hot pixel intersections by starting each ursegment's search from a hot pixel it is known to intersect, and finding all ursegments that have no events but still intersect hot pixels by scanning up and down from segments that do have events.

During each phase, hot pixels are flagged as magnets if they contain at least one ursegment crossing not at the pixel center; otherwise they are pins. The standard snap rounding implementation generates the rounded edges of each ursegment on the fly, remembering the last visited hot pixel for each active ursegment and writing out new edges as new hot pixel intersections are detected. Stable snap rounding extends this by maintaining a list of hot pixels for each active ursegment. The list for ursegment s contains the most recently seen element of $M(s)$, followed by all subsequent pin pixels that s intersects to the left of the sweepline. When a new element $m \in M(s)$ is encountered, the implementation uses the funnel algorithm of Section 3.1 to generate the rounded segments for s's current list, then resets the list to contain the single element m. Because magnet pixels need to be identified only at the current position of the sweepline, a temporary dictionary of magnet y-coordinates, built fresh during each phase, is all the implementation needs.

Test data for the implementation included both synthetic random data and real data from electronic design automation. The random data sets consisted of segments whose endpoints were generated uniformly at random inside a fixed square. The real data sets were generated by extracting polygon edges from an optical process correction (OPC) application. The random data had many intersections per segment ($\Theta(n)$); the real data had relatively few, averaging at most one or two apiece. The

Table 1
Input segments were generated by choosing their endpoint coordinates uniformly from the integers in [0, 999]. Ten sets of input data were generated for each input size on each of the two platforms. Although the number of segment intersections grows quadratically, the output size grows subquadratically, presumably because multiple intersections may occur in each hot pixel. The size of the stable snap rounded output is smaller than that of the snap rounded output by the average value shown in the $\Delta|\text{out}|$ column. Output sizes and run times are given as $\langle\text{mean}\rangle \pm \langle\text{standard deviation}\rangle$. Snap rounding run times are given in seconds. The "SSR/SR" column for each platform shows the average ratio of stable snap rounding run time to snap rounding run time. (I.e., stable snap rounding is faster for values less than 1.) Note that the run time ratios are smaller on Linux than on Solaris, and that the relative cost of stable snap rounding diminishes for larger inputs, for which the implementation overhead of sweepline maintenance consumes a larger fraction of the run time.

|input|	|SR output|	Δ|out|	Linux		Solaris	
			SR time	SSR/SR	SR time	SSR/SR
1000	212 588 ± 6050	475	0.286 ± 0.01	0.98	0.228 ± 0.01	1.11
2000	723 582 ± 14 511	1200	1.113 ± 0.025	0.97	0.912 ± 0.036	1.12
4000	2 080 814 ± 24 304	2089	6.029 ± 0.18	0.97	6.084 ± 0.28	1.04

Table 2
Input segments were extracted from an OPC application. Snap rounding and stable snap rounding were run on each input set, and the snap rounded results were fed back in as input until the result stabilized. Two data sets were used, with 2.2 million and 4.1 million segments, resp. Iterated snap rounding took four rounds to converge for the smaller data set and six rounds for the larger. Sizes and run times are given as in Table 1. Note that when the input has only integral vertices (so that stable snap rounding is a no-op), the run time overhead of stable snap rounding is very small.

Round:	|input|	|SR output|	Δ|out|	Linux		Solaris	
				SR time	SSR/SR	SR time	SSR/SR
1	2 198 933	3300 095	3734	4.02	1.01	4.01	1.06
2	3 300095	3 301 067	972	4.54	0.96	4.29	0.98
3	3 301 067	3 301 073	6	4.53	0.96	4.35	0.97
4	3 301 073	3 301 073	0	4.54	0.96	4.16	1.00
1	4 083 751	8 363 112	69 726	9.90	1.03	9.50	1.09
2	8 363 112	8 375 032	11 920	12.17	0.96	12.15	0.99
3	8 375 032	8 375 804	772	12.35	0.95	12.03	1.00
4	8 375 804	8 375 863	59	12.19	0.96	11.65	1.05
5	8 375 863	8 375 867	4	12.21	0.96	11.72	1.01
6	8 375 867	8 375 867	0	12.36	0.95	11.93	1.01

implementation's output was validated by comparing it with the results of a separate brute-force implementation of stable snap rounding.

The tests show that stable snap rounding is only slightly slower than ordinary snap rounding, on average. The size of the output (measured as the total complexity of all rounded segments, $\sum_{s\in S}|\text{SSR}(s)|$) is generally somewhat smaller than the analogous quantity for ordinary snap rounding. Table 1 shows results for tests on random inputs, and Table 2 shows results for input sets extracted from OPC data. The latter results show how many rounds of ISR may be required to stabilize on real application data. The tests were performed on x86 hardware, using two different OS/compiler platforms, Linux (64-bit RHEL 4, gcc 3.4 compiler, 32-bit object code) and Solaris (64-bit Solaris 10, Sun Studio 11 compiler, 64-bit object code). The two compilers are different enough to affect the relative runtimes by a few percent. In fact, and somewhat counterintuitively, stable snap rounding was often faster than ordinary snap rounding on the Linux platform.

References

[1] J.L. Bentley, T.A. Ottmann, Algorithms for reporting and counting geometric intersections, IEEE Trans. Comput. C-28 (9) (1979) 643–647.
[2] M. de Berg, D. Halperin, M. Overmars, An intersection-sensitive algorithm for snap rounding, Comput. Geom.: Theor. Appl. 36 (2007) 159–165.
[3] W.-K. Chen (Ed.), The VLSI Handbook, 2nd edition, CRC Press LLC, 2006.
[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd edition, MIT Press, Cambridge, MA, 2001.
[5] J.R. Driscoll, N. Sarnak, D.D. Sleator, R.E. Tarjan, Making data structures persistent, J. Comput. Syst. Sci. 38 (1989) 86–124.
[6] H. Edelsbrunner, L.J. Guibas, Topologically sweeping an arrangement, J. Comput. Syst. Sci. 38 (1989) 165–194;
H. Edelsbrunner, L.J. Guibas, Topologically sweeping an arrangement, J. Comput. Syst. Sci. 42 (1991) 249–251 (Corrigendum).
[7] M. Goodrich, L.J. Guibas, J. Hershberger, P. Tanenbaum, Snap rounding line segments efficiently in two and three dimensions, in: Proc. 13th Annu. Sympos. Comput. Geom, 1997, pp. 284–293.
[8] D.H. Greene, Integer line segment intersection, unpublished manuscript.
[9] D.H. Greene, F.F. Yao, Finite-resolution computational geometry, in: Proc. 27th Annu. IEEE Sympos. Found. Comput. Sci., 1986, pp. 143–152.
[10] L. Guibas, J. Hershberger, D. Leven, M. Sharir, R. Tarjan, Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons, Algorithmica 2 (1987) 209–233.
[11] L.J. Guibas, D.H. Marimont, Rounding arrangements dynamically, Int. J. Comput. Geom. Appl. 8 (2) (1998) 157–176.
[12] L.J. Guibas, F.F. Yao, On translating a set of rectangles, in: F.P. Preparata (Ed.), Computational Geometry, in: Adv. Comput. Res., vol. 1, JAI Press, Greenwich, Conn, 1983, pp. 61–77.
[13] D. Halperin, E. Packer, Iterated snap rounding, Comput. Geom.: Theor. Appl. 23 (2002) 209–225.
[14] J. Hershberger, Improved output-sensitive snap rounding, Discr. Comput. Geom. 39 (2008) 298–318.
[15] J.D. Hobby, Practical segment intersection with finite precision output, Comput. Geom. Theor. Appl. 13 (4) (1999) 199–214.
[16] D.T. Lee, F.P. Preparata, Euclidean shortest paths in the presence of rectilinear barriers, Networks 14 (3) (1984) 393–410.
[17] C. Mack, Fundamental Principles of Optical Lithography: The Science of Microfabrication, John Wiley and Sons, 2007.

[18] Mentor Graphics Corp, Calibre nmDRC and eqDRC, http://www.mentor.com/products/ic_nanometer_design/verification-signoff/physical-verification/calibre-nmdrc/.
[19] V. Milenkovic, Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic, in: Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci., 1989, pp. 500–505.
[20] K. Mulmuley, Computational Geometry: An Introduction Through Randomized Algorithms, Prentice Hall, Englewood Cliffs, NJ, 1993.
[21] E. Packer, Iterated snap rounding with bounded drift, Comput. Geom.: Theor. Appl. 40 (3) (2008) 231–251.
[22] A. Ralston, P. Rabinowitz, A First Course in Numerical Analysis, 2nd edition, McGraw–Hill, 1978.
[23] L. Scheffer, L. Lavagno, G. Martin (Eds.), Electronic Design Automation for Integrated Circuits Handbook, Taylor & Francis, Boca Raton, Florida, 2006.