Snap Rounding: A Cautionary Tale

John Hershberger
Siemens EDA, Wilsonville, OR, USA

Abstract

This paper describes the author's experience using and modifying the technique of snap rounding to meet the needs of a circuit verification system. The interplay of theory and practice illuminates the subtle challenges of both.

2012 ACM Subject Classification Theory of computation → Computational geometry; Software and its engineering → Designing software; Applied computing → Computer-aided design

Keywords and phrases Snap rounding, implementation, computational geometry

Digital Object Identifier 10.4230/LIPIcs.SoCG.2025.57

1 Introduction

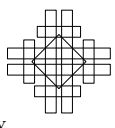
As a discipline, computational geometry is based on real-world problems. Where is the nearest post office? What is the shortest path to grandma's house? What route should a delivery truck follow? To solve these problems, we abstract them, focusing on the key features and ignoring ones that seem irrelevant. We can (we assume) ignore the specifics of post offices, grandparents, and the kinds of goods delivered.

If computational geometry is to remain relevant, the abstract solutions it develops have to be brought to bear on the specific problems that inspired them. The problem features that were ignored may turn out to matter after all. The difference between ice cream and oriental rugs may matter a lot when planning a delivery route! This paper describes one example of applying the computational geometry technique of snap rounding [7, 8, 12] to a problem in the design of integrated circuits.

The specific problem to be solved is computing Boolean operations on polygonal regions. This is a key subroutine in representing the layers of metal, polysilicon, diffusion, etc., used to design a modern IC (integrated circuit) [3, 13, 15]. For example, given a collection of possibly overlapping polygons drawn by a circuit designer, a design system may need to compute the boundary of the union of the polygons. Or alternatively, given two collections of polygons, the system may need to find the points inside the first collection and not the second. (These operations are called single-layer OR and two-layer NOT, respectively.)

Computing the overlay of two subdivisions of the plane is a classic computational geometry problem, with a classic plane sweep algorithm to solve it. From the overlay one can compute any of the Boolean operations simply by noting which input polygons overlap each region of the overlay [5].

Like many computational geometry problems, the overlay problem is stated and solved in the Real RAM model, in which point coordinates are specified and computed with arbitrary precision. On a computer in the real world, the Real RAM is not realistic. Coordinates and computations can use only a limited number of bits. In this problem of polygon Boolean operations, the vertices of the input polygons are specified with k-bit integer coordinates ($k = 64$), and the result polygons must use the same format. This immediately presents a problem of accuracy. If the endpoints of two intersecting line segments are specified with k-bit integers, then rough calculation shows that the intersection point may need up to $5k$ bits to represent each of its coordinates precisely as a rational number. The vertices of the Boolean result must be rounded to the integer grid to satisfy the requirements of the IC design system.




Rounding the result of a geometric computation raises problems of accuracy and topological correctness. How much inaccuracy is introduced by rounding? Is it possible to guarantee that the rounded result faithfully represents the unrounded one? *Snap rounding* was invented to round geometric arrangements as accurately and correctly as possible. Section 2 describes this technique in more detail.

This paper tells the story of my experience using snap rounding in an IC design system. Features of snap rounding that seemed natural and unobjectionable in theory turned out to be ugly stumbling blocks in practice. Resolving these issues led to new theory, further practical challenges, and (eventually) a deeper understanding of the interplay between theory and practice.

2 Definitions

Snap rounding solves the problem of rounding the vertices of an arrangement of line segments to the integer grid. The algorithm is simple to define and relatively simple to implement; it comes with guarantees on the quality of its results [2, 6, 7, 8, 9, 10, 12, 14].

The input to snap rounding is a collection of *ursegments* ("ur" is short for "UnRounded" and is also a German prefix meaning "original"). In the IC design application, ursegment endpoints are 64-bit integer grid points, but snap rounding itself does not restrict its inputs. The vertices of the arrangements of ursegments, both endpoints and intersections, determine the rounding. The Cartesian plane is divided into *pixels*, unit squares centered on the integer grid points. Pixels are closed on their left and bottom sides and open on their right and top sides – thus for integer x and y, the corner $(x - \frac{1}{2}, y - \frac{1}{2})$ belongs to the pixel centered at (x, y), and the other three corners do not. Snap rounding is defined by the interaction of ursegments and pixels:

> Every pixel that contains a vertex of the ursegment arrangement becomes a *hot pixel*. Every ursegment s is replaced by a polygonal path joining the centers of the hot pixels that s intersects.

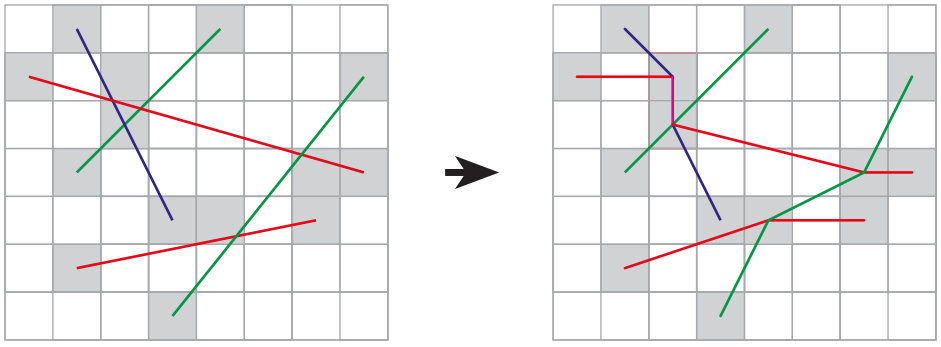


Figure 1 Hot pixels are created by vertices of the ursegment arrangement. Each ursegment is replaced by a polygonal path connecting hot pixel centers.

This simple rounding scheme, illustrated in Figure 1, has two desirable features, as proven by Guibas and Marimont [8]: no segment moves by more than half a pixel, and the topology of the input arrangement is preserved up to collapsing of features. (Here the word "topology" is used to refer to adjacency and ordering relationships.) The order of intersections between ursegments is preserved in the rounded segments, the order of intersection of ursegments with a vertical or horizontal line is preserved, and triangle orientations are preserved, all up

to collapsing of features. This reflects the fact that the rounded arrangement is a homotopic deformation of the arrangement of ursegments. The proofs are sketched below because they will be needed later.

▶ **Lemma 1.** *No segment moves by more than half a pixel. That is, the L_∞ Hausdorff distance between any ursegment and its rounded representation is at most $\frac{1}{2}$.*

Proof. For every ursegment $s = \overline{ab}$, we pick a point $p_i \in s$ belonging to each hot pixel that s intersects, with the first and last points equal to the segment endpoints: $a = p_1, \ldots, p_k = b$. Thus s is trivially represented by the polygonal path $(p_1, \ldots, p_k)$. Let q_i denote the center of the hot pixel containing each p_i, and define $\Delta_i = p_i - q_i$. Since p_i belongs to the hot pixel centered at q_i, $||\Delta_i||_\infty \leq \frac{1}{2}$. The rounded version of s is the polygonal path $(q_1, \ldots, q_k)$. Any point $p \in \overline{p_i p_{i+1}}$ is a convex combination of the endpoints, $p(t) = (1-t) \cdot p_i + t \cdot p_{i+1}$, for $t \in [0, 1]$. The corresponding point of $\overline{q_i q_{i+1}}$ is $q(t) = (1-t) \cdot q_i + t \cdot q_{i+1}$. By linearity, $p(t) = q(t) + ((1-t) \cdot \Delta_i + t \cdot \Delta_{i+1})$, so $||p(t) - q(t)||_\infty \leq \frac{1}{2}$. This one-to-one mapping between points of s and points of the rounded segment shows that the Hausdorff distance between s and the rounded segment is at most $\frac{1}{2}$. See Figure 2. ◀

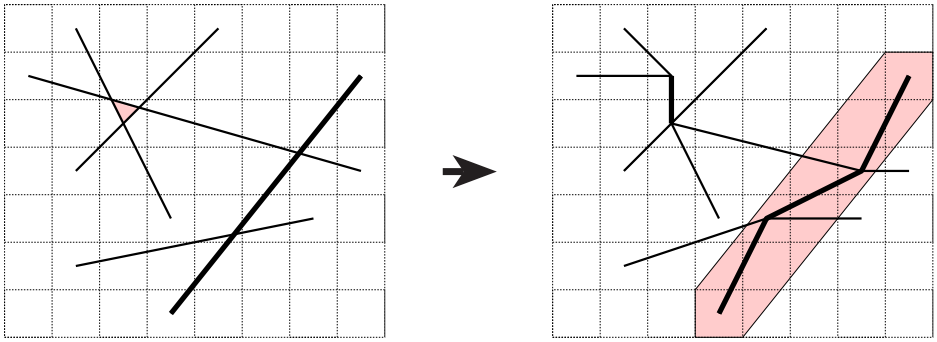


Figure 2 The rounded version of the lower right ursegment is contained in the region within L_∞ distance $\frac{1}{2}$ of the ursegment.

▶ **Lemma 2.** *The topology of the arrangement of ursegments is preserved in the rounded arrangement, up to collapsing of features.*

Proof. (Summary) Guibas and Marimont's proof uses a continuous deformation of the arrangement that maps the input arrangement to the rounded one. Topology is preserved at every instant of the deformation, and hence it is preserved over the whole deformation.

The first step of the argument introduces internal vertices on each ursegment at its first and last contacts with hot pixel boundaries. The deformation shrinks each hot pixel to its center, first by a uniform horizontal contraction to a line segment and then by a vertical contraction to a point. All the hot pixels shrink simultaneously, at the same rate. The arrangement inside each hot pixel shrinks uniformly (affinely) with the hot pixel, so all spatial and topological relationships inside the pixel are preserved (up to collapsing of features). As the hot pixels shrink, the segments outside the hot pixels move, dragged along by their endpoints on the hot pixel boundaries. Guibas and Marimont's key observation is that no external segment can pass over another segment or vertex without crossing into a hot pixel (because all endpoints and ursegment crossings are inside hot pixels), but the deformation is defined such that the hot pixel corners move away from the segments at least as fast as the segments approach – the segments never catch up with the corners.

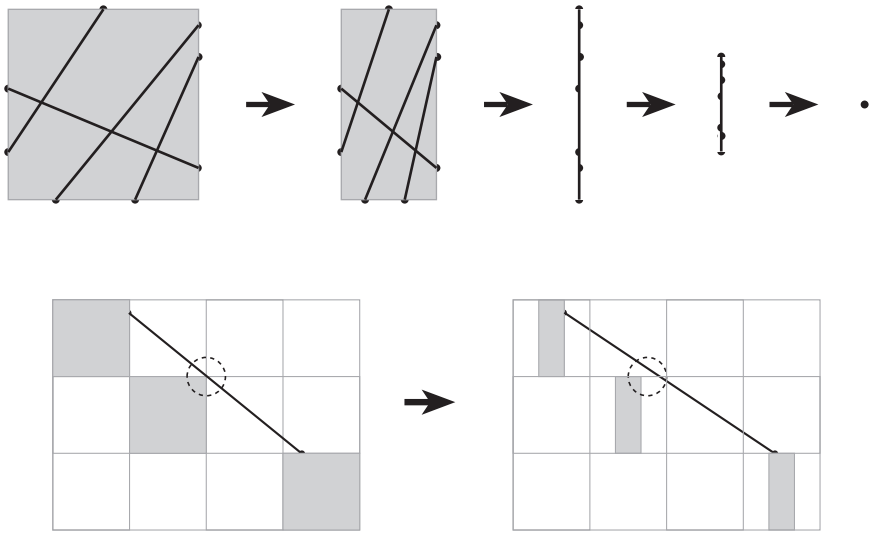


Figure 3 Contraction of a hot pixel does not affect topological relationships inside it. A deforming edge that starts outside all hot pixels never crosses into any hot pixel.

Specifically, let p_0 and p_1 be two consecutive breakpoints along some ursegment, and let $p(\tau)$ be a point on the segment between them, with $p(\tau) = (1-\tau) \cdot p_0 + \tau \cdot p_1$ for $\tau \in [0,1]$. During the horizontal contraction phase, let $p_0(t)$, $p_1(t)$, and $p(\tau,t)$ be the corresponding points at time t of the contraction, where t is the hot pixel half-width, decreasing uniformly from $\frac{1}{2}$ to 0. As t decreases, the corners of all the hot pixels move inward (left or right) at speed 1. Endpoints $p_0(t)$ and $p_1(t)$ move left or right at speed at most 1 (exactly 1 if the endpoint is on the left or right side of a hot pixel; less if it is on the bottom or top). It follows by linearity that $p(\tau,t)$ also moves left or right at speed at most 1, and therefore, because $p(\tau)$ starts outside all hot pixels, $p(\tau,t)$ never crosses into the interior of one. The same argument also works for the vertical contraction phase. See Figure 3. ◀

3 Theory meets practice

I implemented snap rounding as part of a C++ application for physical verification of integrated circuits. Because I wanted to apply snap rounding to solve a variety of problems, I used C++ templates to separate the rounding algorithm from the applications that use it in a scanline sweep.

The scanline package is partitioned into six policies, each with its own role and interface. The *input policy* extracts polygon edge data from a compressed in-memory format and presents them in left-to-right, bottom-to-top order; the *scanline representation* holds the edges intersected by the scanline at its current x-position; the *rounding policy* receives edge events in sorted order and maintains the scanline representation while generating events that describe the rounded subsegments; the *region characteristic* maintains an application-specific characterization of the region above each edge in the scanline (e.g., the depth of coverage); the *output policy* receives rounded segments and consults the scanline representation and region characteristics to produce application-specific output (e.g., polygon Boolean results); and the *event manager* coordinates the interactions between all the other subsystems, including maintaining event queues for the rounding policy and updating the region characteristics

exactly where they need to be recalculated. This decomposition of scanline functionality into distinct roles allows each subsystem to be upgraded independently of the others and allows for substantial reuse between applications. Most applications can use off-the-shelf versions of the input policy, scanline representation, rounding policy, and event manager. The region characteristic can often be reused. Only the output policy always needs a custom implementation.

This structure is both a strength and a weakness of the scanline package. It allows snap rounding to be deployed in a variety of applications with very good reliability, but the strict adherence to interface communication protocols means that out-of-band communication between subsystems is difficult or impossible, say if one subsystem needs to compensate for a problem in another.

Note that application design has already influenced the choice of algorithm. I wanted to use snap rounding to find incidences between polygons and/or edges, as one of the output policies, so the rounding policy needs to identify all the ursegments that contribute to each rounded segment. (Multiple ursegments may be rounded to the same segment in the rounded arrangement.) That means the rounding policy must provide more information than is present in the rounded arrangement of ursegments. That is, the number of notifications is asymptotically larger than the size of the rounded arrangement in the worst case. See Figure 4. Hershberger proposed an output-sensitive snap rounding algorithm that deals with these configurations efficiently [10], but the algorithm is more complicated than necessary for the inputs that arise in practice.

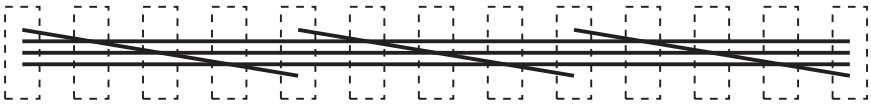


Figure 4 A pathological example with $O(n^2)$ segments in the rounded arrangement, each with $O(n)$ ursegments contributing (rounding) to it. Pixels are stretched vertically for clarity.

The implementation of the rounding policy uses a simple Bentley-Ottmann scanline [1]. The algorithm sweeps a vertical line over the arrangement of ursegments from left to right, maintaining the intersection of the line with the arrangement at each position. Whenever an ursegment intersection or endpoint is detected, it makes the containing pixel *hot*. A local search in the scanline finds all the ursegments that intersect each hot pixel.

This high-level picture becomes a bit more complicated when we look at the details. The implementation gains efficiency by exploiting the knowledge that the result of the Bentley-Ottmann scan is a rounded arrangement. Unlike a classic Bentley-Ottmann scanline, this one does not need to know the precise relative x-order of the ursegment intersections. It is enough to identify the hot pixel containing each intersection and determine whether the intersection lies in the left or right half of the pixel. The crossings inside each half-pixel can then be processed in arbitrary order. Ignoring the precise x-ordering of crossing events reduces the polynomial degree of the required arithmetic computations from quintic to cubic.

Note that the Bentley-Ottmann scanline is a suboptimal algorithmic choice, at least in theory. The scanline processes every vertex of the input arrangement $\mathcal{A}_{in}$, giving complexity $O(|\mathcal{A}_{in}| \log n)$. But $|\mathcal{A}_{in}|$ may be $\Omega(n^2)$ while the output complexity $|\mathcal{A}_{out}|$ is $O(n)$, as shown in Figure 5. It is possible to round such an arrangement in $O(n \log n)$ time, using an algorithm that spends $O(\log n)$ time for each incidence between an ursegment and a hot pixel [6]. However, this is unimportant in the IC design application: arrangements like the one in Figure 5 do not occur in practice. Simplicity of implementation beats theoretical worst-case efficiency.

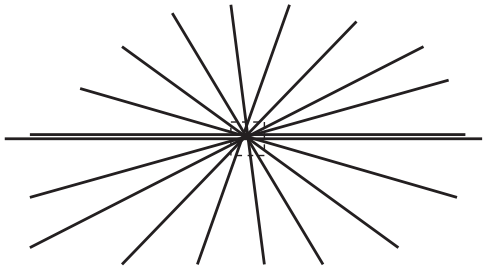


Figure 5 $O(n^2)$ ursegment intersections occur inside a single hot pixel.

The output policy for Boolean operations on polygons maintains a "depth of coverage" (winding number) above each segment in the scanline. Every edge is assigned a value of $+1$ or -1, depending on whether the polygon interior it bounds lies above or below it. The sum of the edge values along the scanline from $-\infty$ to each point on the scanline determines how many polygons cover that point. In the OR operation, for example, a region of the rounded arrangement of segments belongs to the result if its depth of coverage is positive. When the rounding policy notifies the event manager of the left/right end of a rounded segment, it also provides a pointer into the scanline. The event manager uses these pointers to update the depth of coverage counts exactly where they have been changed by scanline updates. When the right end of a rounded segment is processed, the segment gets written to the result by the output policy if and only if the segment forms the boundary between a region inside and a region outside the desired Boolean result. Because of rounding, two segments that are classified as being on the result boundary by depth of coverage may snap to be collinear. When that happens, the two segments annihilate – both are removed from the result. Correspondingly, two regions of the result interior are separated or joined, depending on whether the result interior lies between the two annihilating edges or not.

The design system requires each connected polygon to have a unique polygon number. Here "connected" refers to the polygon interior – polygons may be nonsimple (have holes), but are not connected by contact at vertices. Polygon numbers are assigned on the fly using a disjoint set (Union-Find) data structure [4].[1] Each edge in the output gets its own number; the sets for two edge numbers are joined if the edges are adjacent on the boundary of the result, or if they are vertically visible through the result interior. (Both of these conditions can be checked in constant time per edge.) At the end of the scan, the disjoint set structure is condensed to map the edge numbers to a dense set of polygon numbers. One pass through the output then suffices to replace every edge number with its corresponding polygon number.

The implementation of Boolean operations based on snap rounding replaced an earlier implementation that used a heuristic rounding scheme. That algorithm also detected intersections with a scanline, breaking each segment at rounded intersections. However, the right-hand side of each broken segment was subject to future rounding, so it was possible, though unlikely, for rounded segments to drift arbitrarily far. A second weakness became apparent in the use of Boolean operations to partition and reconstruct polygonal regions in

[1] The disjoint set structure is a simple array, with all joins linking an element to a predecessor. That means that height-balancing (linking the shorter tree to the taller) is not possible, and hence the theoretical performance bound of $O(n\alpha(n))$ is not achievable. However, path compression gives an $O(n \log n)$ worst-case performance bound, and performance in practice is excellent.

IC design. When a polygon is fractured into pieces, geometric rounding may leave tiny gaps (less than a pixel wide) between them. It is desirable for the OR of the pieces to fill in those gaps, but the previous algorithm was inconsistent in achieving that goal.

The new implementation of Boolean operations based on snap rounding has proven to be very successful. Performance is generally good, and snap rounding gives much more predictable, dependable results than the previous heuristic algorithm. Lemma 1 guarantees that rounded results are within half a pixel of the input ursegments – no segment drift is possible. Lemma 2 guarantees topological correctness. Furthermore, the fact that snap rounding moves all ursegments that hit a hot pixel is also beneficial: when snap rounding computes the OR of polygons with tiny gaps between them, the result heals up the gaps better than the earlier implementation.

4 Practice meets theory

The trouble with abstracting a practical problem to get something clean and simple is that some goals may get lost in the translation from practice to theory. This isn't an issue for basic problems like sorting, but becomes relevant for more complex problems. For example, in the problem of rounding an arrangement, symmetry is a practical goal that is difficult to achieve.[2]

In snap rounding, every ursegment that passes through a hot pixel is deformed to pass through the center of the hot pixel. But what if an ursegment merely grazes a hot pixel? What counts as "passing through"? A hot pixel is defined to be closed on its left and bottom sides, and open on its top and right sides. Thus out of the four corners of a hot pixel, only the lower left belongs to it. This means that for the configuration of segments on the left side of Figure 6, the rounded result is asymmetric, as shown on the right.

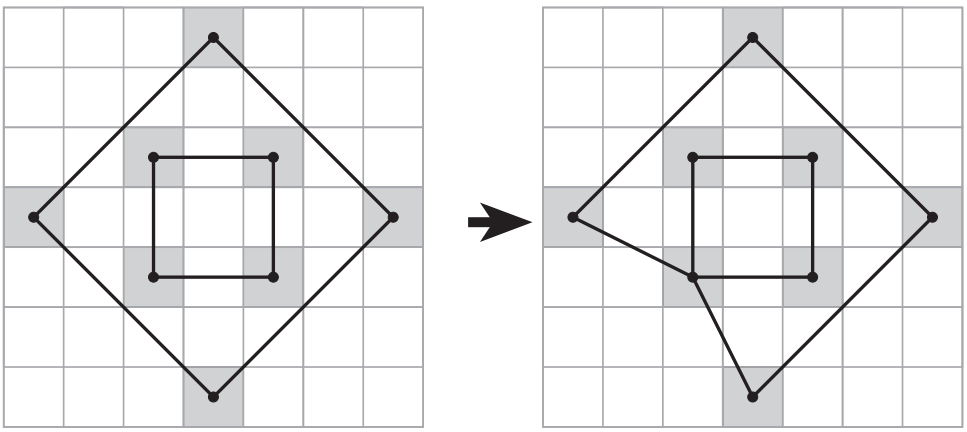


Figure 6 Snap rounding deforms one of the four segments of the outer diamond to pass through a vertex of the inner square.

[2] For efficiency's sake, modern ICs are designed hierarchically. Smaller "cells" (design components) are placed within larger cells. The result of instantiating all the placements, all the way down, is the *flat* design, which should be equivalent to the hierarchical design. However, in a hierarchical design, geometric rounding is performed in the frame of reference of individual cells. If those cells are rotated by a multiple of 90° before placement, as often occurs, any asymmetry in rounding creates a difference between the hierarchical design and the ideal flat design, in which rounding occurs after expansion of placements. Thus asymmetry is to be avoided if possible.

The previous rounding algorithm, the one that snap rounding replaced, did not perturb any of the segments in the left-hand configuration. Thus for this configuration and others like it, the results of snap rounding are perceived as worse by the users of the polygon Boolean application. Unfortunately, predictability and solid definitions are virtues that appeal to theoreticians more than to practitioners. The practitioner is likely to say, "The new algorithm gives worse results on this example. I want an algorithm that does *the right thing* on all of my test cases." The theoretician can say, loftily, "This undisciplined wish for a simultaneous solution to incompatible requirements is how we ended up with the original mess of an algorithm," but it's a tough argument to make.

5 Relaxed snap rounding (Theory gives in to practice)

Eventually, the complaints of the practitioners wore me down. Yes, by the rules of snap rounding, the rounded version of the configuration in Figure 6 has to be asymmetric, but perhaps there's *another* rounding scheme that would give better results.

For example, Hershberger's *stable snap rounding* algorithm might fit the bill [11]. Stable snap rounding has two kinds of hot pixels, magnets and pins, with magnets created by crossings or vertices not at the pixel center. An ursegment that hits a magnet is forced to pass through its center, but one that hits a pin is not – if the segment moves because of magnet vertices, the only restriction is that it may not pass over a pin, as illustrated in Figure 7. In the example of Figure 6, stable snap rounding would not move any of the segments, leaving a symmetric result.

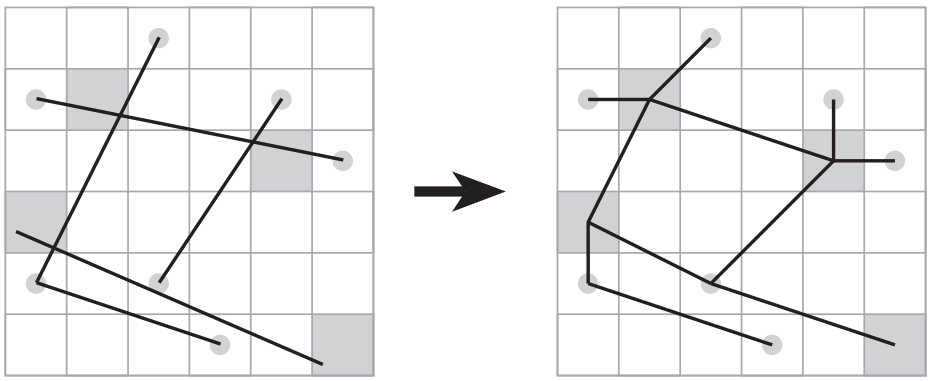


Figure 7 Shaded squares are magnets (pixels containing ursegment crossings or off-grid endpoints); shaded disks are pins (all other hot pixels). Stable snap rounding does not force ursegments to deform to pins, but maintains ursegment homotopy relative to pins.

I didn't want to implement stable snap rounding to solve the asymmetry problem. The behavior of stable snap rounding is much different from ordinary snap rounding. It was hard enough for our software testers to adapt to the differences between snap rounding and the previous algorithm – I didn't want to start the whole qualification process over again. Instead, I wanted a quick, small change that would fix this little asymmetry problem without breaking anything else. [Reader, if you are feeling uneasy about this goal – you have cause!]

I came up with the idea of *relaxed snap rounding*. This borrows an idea from stable snap rounding and applies it in one special circumstance. The rules of relaxed snap rounding are the same as those of ordinary snap rounding *except*

> If a hot pixel contains no ursegment intersections and has ursegment endpoints only at its center, then an ursegment that crosses the pixel's lower left corner *is not considered to contact the hot pixel*.

That is, if a segment crosses the lower left corner of a pixel that is hot only because of ursegment endpoints at its center, the segment is not required to deform to the pixel's center. All other rules remain the same – if an ursegment touches a hot pixel's interior, it *does* deform to the pixel center.

I was quite pleased when I came up with the idea of relaxed snap rounding. In fact, I even considered writing a short paper about it. Relaxed snap rounding solved the asymmetry problem, and I could prove that it had the same good properties as ordinary snap rounding.

- No ursegment moves by more than half a pixel. The original argument applies unchanged. An ursegment is deformed to pass through a subset of the hot pixels that it touches under ordinary snap rounding, so *a fortiori* its deformation is no greater than that produced by ordinary snap rounding.
- Topology of the input arrangement is preserved up to collapsing of features. This, too, can be proved by an adaptation of the original argument of Guibas and Marimont. Once again, we break each ursegment at its intersections with hot pixel boundaries, except that for each pixel made hot only by ursegment endpoints at its center, any segment passing only through its lower left corner is not broken there. Each hot pixel is collapsed to a point, first by uniform horizontal contraction of all hot pixels simultaneously, then by uniform vertical contraction of all the hot pixels (which are now just vertical segments). Each ursegment is split into subsegments at its breakpoints on the hot pixel boundaries, and as the hot pixels contract, the subsegments move with their breakpoint vertices. Every internal point of a subsegment is a convex combination of the subsegment endpoints. Since each endpoint moves left or right (during the first contraction) or up or down (during the second) with speed at most 1, the same is true of every internal point of a segment. Because the corners of each hot pixel move inward at speed 1, no subsegment that starts strictly outside all hot pixels can ever enter any hot pixel. Since all ursegment intersections and endpoints are inside hot pixels, the original arrangement's topology is preserved. In the worst case, a segment that passes through a hot pixel's lower left corner may stay in contact with that corner as it deforms, so the segment ends up passing through the hot pixel center. However, this still preserves topology up to collapsing of features.

Relaxed snap rounding was a big success! The asymmetry problem was resolved, with very few changes to behavior overall. It was rare for an ursegment to pass through the lower left corner of an endpoint-only hot pixel, and when the situation did occur, leaving the segment unbroken at that corner gave more desirable results than the alternative.

6 The fly in the ointment

It was not until a year later that the other shoe dropped. People began to notice that the polygon Boolean operations sometimes produced results that were not well-formed polygons. Some polygons were only weakly simple, like the one shown in Figure 8.

What was the cause of these errors? After several days of debugging, eventually I traced the problem back to a previously overlooked weakness in relaxed snap rounding: no snapping means no noticing. The output policy gets notified of a snapped segment at the segment's right end, when the segment is snapped to pass through a hot pixel's center. The polygon

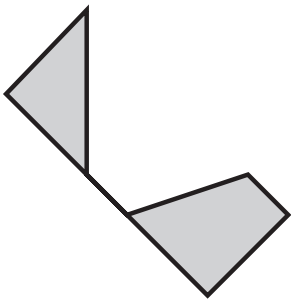


Figure 8 Relaxed snap rounding created polygons with disconnected interiors!

Boolean operation uses that notification to identify oppositely-oriented segments that cancel each other out, and to identify vertices where a polygon pinches off into two. With relaxed snap rounding, a snapped segment may pass through a hot pixel center, but the incidence is not noticed when the hot pixel is processed. The incidence is caused by snapping in another, later hot pixel. See Figure 9.

Note that this problem does not occur for ordinary snap rounding:

▶ **Lemma 3.** *Under ordinary snap rounding, an ursegment s is snapped to pass through the center of a hot pixel h if and only if s contacts h.*

Proof. Contact implies snapping, so the only thing to prove is that if s does not contact h, its snapped version cannot pass through the center of h.

Any point on s moves by at most $\frac{1}{2}$ in x and/or y during the snap rounding deformation, which means that any point of s strictly outside the closure of h (denoted $\overline{h}$) cannot deform to the center of h. The points of $\overline{h} \setminus h$ are the top and right edges and the three corners incident to those edges. If a point $p \in s$ is on the top edge of h and internal to that edge, then s is horizontal – otherwise it would intersect the interior of h. In this case all the hot pixels that s contacts are above h, so s will snap upward, away from h. A similar argument applies for s collinear with the right edge of h. If s passes through a non-contact corner of h (denoted by p) and deforms to pass through the center of h, let h_1 and h_2 be the hot pixels that s contacts immediately before and after p. For p to move to the center of h, the intersections of s with $\overline{h_1}$ and $\overline{h_2}$ must both move by $\frac{1}{2}$ in both x- and y-coordinates, in the same direction as from p toward the center of h. This implies that the contact between s and h_1 (resp. h_2) is at the same corner of h_1 (h_2) that p occupies relative to h. But contact is defined the same for all hot pixels, so since s does not contact h at p, it cannot contact h_1 or h_2, either, a contradiction. ◀

That is to say, in a scanline implementation of ordinary snap rounding, every incidence between a snapped ursegment and a hot pixel center is detected when the hot pixel is processed, but this is not true for relaxed snap rounding. As in Figure 9, it is possible for polygon Boolean operations using relaxed snap rounding to produce a pinch-off, e.g., to separate a single result polygon into two components, without detecting it.

7 The way forward

In hindsight, relaxed snap rounding seemed like a big mistake. It sacrificed the algorithmic guarantee of ordinary snap rounding for a small improvement in result quality. Unfortunately, once user expectations had been set for relaxed snap rounding's behavioral improvements, it was impossible to go backward.

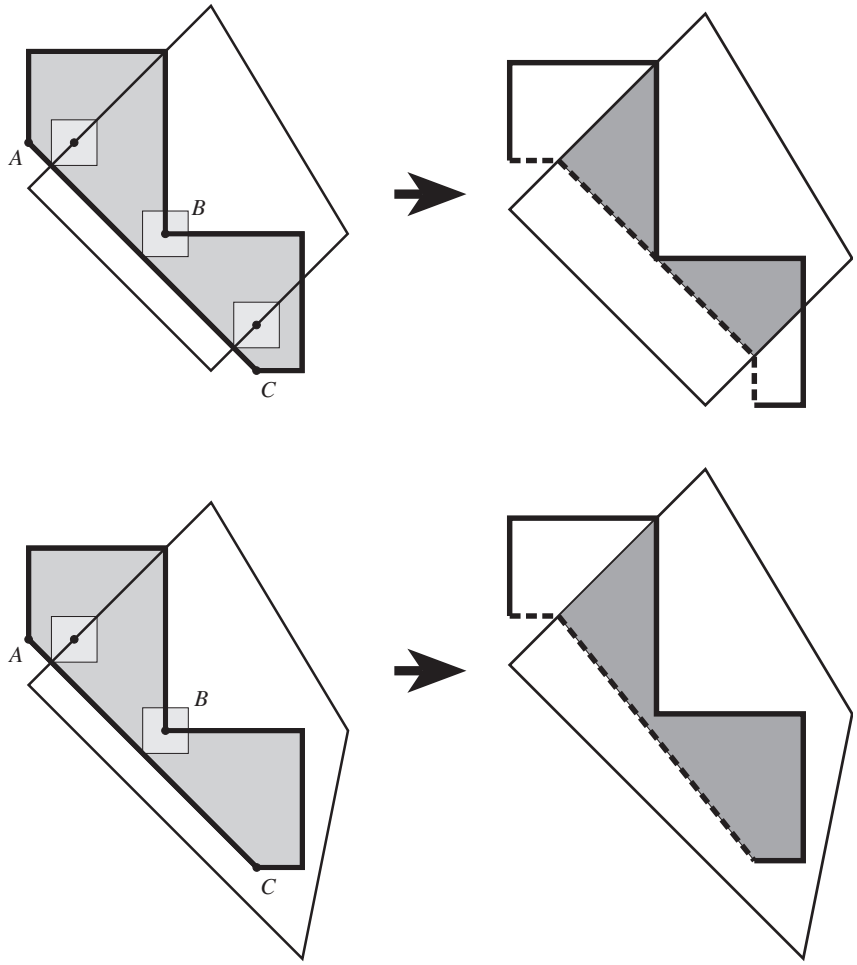


Figure 9 The shaded polygon is ANDed with the unshaded polygon. Depending on what occurs to the right of B, the segment $\overline{AC}$ may or may not snap to pass through B, creating either two or one polygons in the AND result. The relevant hot pixels are highlighted. The incidence between $\overline{AC}$ and the hot pixel at B in the upper figure is not detected by relaxed snap rounding. The snapping that creates the incidence does not occur until the scanline is past B. When the scanline is at position B, the configurations in the upper and lower figures are indistinguishable, even though the results of the AND near B are very different.

I had to find a way to reconstruct the incidences that relaxed snap rounding fails to detect. In this case, the separation of the rounding policy from the polygon Boolean output policy is a liability. The rounding policy knows when a contact between an endpoint-only hot pixel and an ursegment is being ignored, but "ignoring" means that the rounding policy does not generate an end-of-segment event for the non-incidence. The output policy is left to compensate for the rounding policy's omissions.

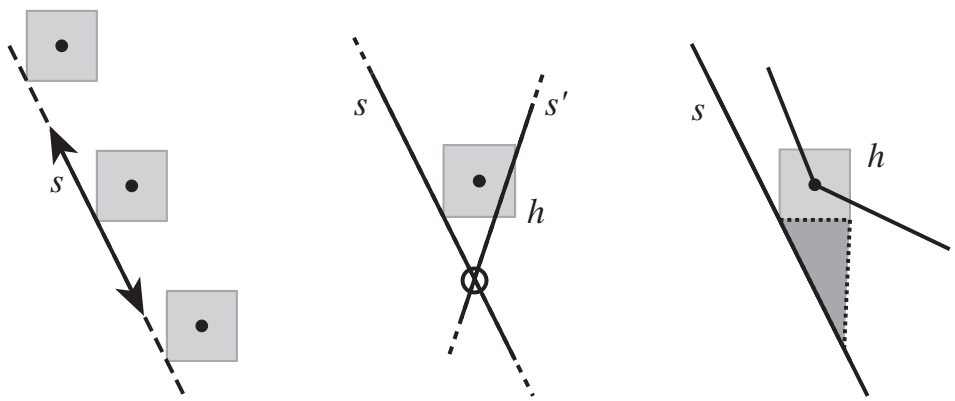
As a first step, I defined a *vulnerable vertex* as a hot pixel that contains no crossings and has an ursegment passing through its lower left corner. Vulnerable vertices can lead to undetected incidences, and every undetected incidence occurs at a vulnerable vertex. The following lemma characterizes these vulnerable vertices.

▶ **Lemma 4.** *Let s be an ursegment that passes through the lower left corner of a hot pixel h without touching the interior of h. If the contact between s and h is ignored under relaxed snap rounding, but s is snapped to pass through the center of h, then*

1. *The x-span of s contains the x-span of h in its interior.*
2. *If $s' \neq s$ is an ursegment that extends to the left of the x-span of h and rounds to pass through its center, then its leftmost intersection with h is on the top or left boundary.*
3. *If any ursegment $s' \neq s$ extends to the left of h and rounds to pass through its center, then s is immediately below the lowest such segment. Otherwise, s is immediately below the lowest segment with its left endpoint in h.*

Proof. Note that because s's contact with h is ignored, h has at least one ursegment vertex at its center, but contains no ursegment intersections.

(1) Let h_l and h_r be the hot pixels that s contacts before and after h. Since s snaps to pass through the center of h, s needs to move by $(\Delta x, \Delta y) = (\frac{1}{2}, \frac{1}{2})$ at both h_l and h_r, i.e., it must pass through their lower left corners. That means h_l must be strictly left of h, and h_r must be strictly to the right. See Figure 10(1).



(1) (2) (3)

Figure 10 (1) The hot pixels that s contacts before and after h are strictly left and right of h. (2) Ursegments s and s' intersect in the column of pixels below h. (3) The shaded triangle below h must be empty of ursegments (except possibly for ones that contact h).

(2) Let x_h be the x-coordinate of h's center. If s' first intersects the bottom of h, then it has positive slope. Segment s has negative slope. Consequently s and s' must intersect in the x-interval $(x_h - \frac{1}{2}, x_h + \frac{1}{2})$, somewhere below h. This means that h_r, the first hot pixel on s right of $x_h - \frac{1}{2}$, has its center on $x = x_h$. By (1), this means that s cannot snap to pass through the center of h. See Figure 10(2).

(3) If any ursegment that does not contact h intersects the interior of the triangle formed by s, the vertical line $x = x_h + \frac{1}{2}$, and the bottom of h, then either s has an intersection in the x-interval $(x_h - \frac{1}{2}, x_h + \frac{1}{2})$ or it crosses a hot pixel created by an ursegment endpoint in the column below h. In either case, h_r is not to the right of h, contradicting (1). Hence this triangle is free of all ursegments except possibly for ones that intersect h, implying that s is immediately below the lowest left or right segment in h, as claimed. See Figure 10(3). ◀

It is possible to detect vulnerable vertices in constant time apiece using the events that *are* passed to the output policy. Whenever the scanline passes the left or right end of an ursegment, it generates an event that is reported to the output policy with a pointer to the scanline location where the vertex occurs. Any segment with an undetected incidence lies just below one of the segments with a vertex at h's center, by Lemma 4(3). By looking at the neighboring edges in the scanline for each endpoint event, it is possible to identify all vulnerable vertices.

I used vulnerable vertices as the basis of a two-part scheme to restore correct behavior for Boolean operations. The first step uses vulnerable vertices to identify undetected incidences. The vulnerable vertices are identified in left-to-right order and stored in a table in that order. The detection scheme keeps a sliding window of vulnerable vertices. The width of the window is defined by the edges intersecting the current scanline. When the scanline reaches the right end of a snapped segment s that forms part of the result polygon, if s snaps up and to the right by $(\frac{1}{2}, \frac{1}{2})$, the scheme searches the active vulnerable vertices to see whether the snapping creates an incidence in the segment interior. If so, the identity of the output polygon containing s is recorded for later processing.

The second part of the scheme processes any incidences that have been discovered. Fortunately, incidences occur rarely enough in practice that fairly expensive cleanup is affordable. After the polygon Boolean operation completes, if any internal incidences were detected, the cleanup code extracts the edges of all polygons containing an incidence and performs an OR operation again on the extracted edges, without further snapping. Then these updated polygons are renumbered and integrated back into the rest of the output. This, finally, restores correct output.

This is quite ugly and potentially inefficient, as the reader will have observed. With the addition of more data structures, it could be made more efficient, but for now, it seems to be efficient enough for the application. I declared victory and slunk from the field.

8 Lessons learned

The most obvious take-away from this experience is no surprise at all: implementation is hard, and we theoreticians ignore that fact, or sweep it under the rug, at our peril. The requirements of an application often do not precisely match the assumptions of a theoretical algorithm. My mistake in "proving" the correctness of relaxed snap rounding was that I forgot the dependencies of the polygon Boolean operation on the scanline implementation of snap rounding. Although relaxed snap rounding meets its approximation guarantees, it fails to detect all incidences between snapped segments and hot pixel centers, and the polygon Boolean operation depends on incidence detection for correctness. Even if incidence detection is restored with a post-processing step, incidences are not detected in left-to-right order, so the simplicity of the scanline Boolean algorithm is compromised.

As someone who has worked in both the theoretical and applied communities for many years, I think that in some ways, the life of a theoretician is easy – if the problem as posed is too hard, we can solve a related problem and wave our hands to argue that the new problem is just as interesting as the original. The life of an implementor of applications is easy, too – there's no need for an optimal implementation, just one that is correct (enough) and fast enough on the data that occur in practice. However, the life of someone who wears both hats can be very challenging: finding (or inventing) an algorithm that solves the given problem efficiently and is simple enough to implement is hard to do repeatedly. Nevertheless, success, even only occasional success, is quite satisfying when it comes, and makes the struggle worthwhile.

That brings me to the key question of this section, "Have *I* learned *my* lesson?" For good or ill, the answer is *no*. I continue to struggle (actually, to play) at the boundary between theory and practice, trying to find or make theory that is practical enough to use, trying to find practical problems that are elegant enough to appeal to theoreticians, and doing my best to implement elegant, efficient solutions to real-world problems.

References

1 J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979. doi:10.1109/TC.1979.1675432.

2 M. de Berg, D. Halperin, and M. Overmars. An intersection-sensitive algorithm for snap rounding. *Comp. Geom.: Theory and Appl.*, 36:159–165, 2007. doi:10.1016/J.COMGEO.2006.03.002.

3 W.-K. Chen, editor. *The VLSI Handbook*. CRC Press LLC, 2nd edition, 2006.

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

5 M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

6 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.

7 D. H. Greene. Integer line segment intersection. Unpublished Manuscript.

8 L. J. Guibas and D. H. Marimont. Rounding arrangements dynamically. *Int. J. Comp. Geometry & Appl.*, 8(2):157–176, 1998. doi:10.1142/S0218195998000096.

9 D. Halperin and E. Packer. Iterated snap rounding. *Comp. Geom.: Theory and Appl.*, 23:209–225, 2002. doi:10.1016/S0925-7721(01)00064-5.

10 J. Hershberger. Improved output-sensitive snap rounding. *Discr. Comput. Geom.*, 39:298–318, 2008. doi:10.1007/S00454-007-9015-0.

11 J. Hershberger. Stable snap rounding. *Comp. Geom.: Theory and Appl.*, 46(4):403–416, 2013. http://dx.doi.org/10.1016/j.comgeo.2012.02.011. doi:10.1016/J.COMGEO.2012.02.011.

12 J. D. Hobby. Practical segment intersection with finite precision output. *Comput. Geom. Theory Appl.*, 13(4):199–214, October 1999. doi:10.1016/S0925-7721(99)00021-8.

13 C. Mack. *Fundamental Principles of Optical Lithography: The Science of Microfabrication*. John Wiley and Sons, 2007.

14 E. Packer. Iterated snap rounding with bounded drift. *Comp. Geom.: Theory and Appl.*, 40(3):231–251, 2008. doi:10.1016/J.COMGEO.2007.09.002.

15 L. Scheffer, L. Lavagno, and G. Martin, editors. *Electronic Design Automation for Integrated Circuits Handbook*. Taylor & Francis, Boca Raton, Florida, 2006.