Efficient Snap Rounding with Integer Arithmetic

Binay K. Bhattacharya* Jeff Sember†

Abstract

In this paper we present a slightly modified definition of snap rounding, and provide two efficient algorithms that perform this rounding. The first algorithm takes n line segments as input and generates the set of snapped segments in $O(|I| + \Sigma_c\ is(c) \log n + |I^*_m|)$, where $|I|$ is the complexity of the unrounded arrangement I, $is(c)$ is the number of segments that have an intersection or endpoint in pixel column c, and I^*_m is the multiset of snapped segment fragments. The second algorithm generates the rounded arrangement of segments in $O(|I| + \Sigma_c\ is(c) \log n + |I^*| \log n)$, where $|I^*|$ is the complexity of the rounded arrangement I^*. Both use simple integer arithmetic to compute the rounded arrangement by sweeping a strip of unit width through the arrangement, are robust, and are practical to implement. They improve upon existing algorithms, since existing running times either include a logarithmic factor in $|I|$, (i.e., $|I|\ \log n$), or depend upon the number of segments interacting within a particular hot pixel ($is(h)$ and $ed(h)$ [7], or $|h|$ [3]), whereas ours are linear in $|I|$ and depend upon the number of segments interacting in an entire hot column ($is(c)$), which is a much coarser partition of the plane.

1 Introduction

Snap rounding is a method for transforming an arbitrary-precision arrangement of segments to a fixed-precision representation, while attempting to retain certain features of its geometry and topology.

We are given a set $S = \{s_1, \ldots, s_n\}$ of line segments in $\mathbb{R}^2$. We wish to round the arrangement I of S to a grid of pixels. In this note, we assume that segment endpoints are integers (though our algorithms can handle endpoints that are rational numbers), and each pixel is centered at a point with integer coordinates. In snap rounding, each pixel containing a vertex in I is 'hot', and segments intersecting any hot pixel are rerouted to pass through the pixel's center (a process we call *snapping the segment to a pixel*); see figure 1.

We refer to each original (unrounded) line segment as an *ursegment*, and the polygonal line resulting from its being snap rounded a *polysegment*. Each polysegment is comprised of *fragments*.

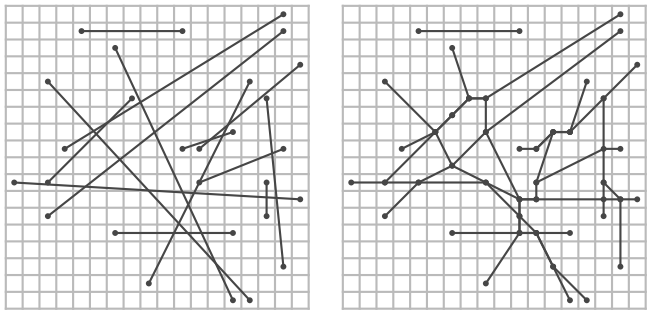

Figure 1: Snap rounding a set of line segments.

Snap rounding was introduced independently by Hobby [8] and Greene [4]. An algorithm with running time $O(n \log n + \Sigma_{h \in H} |h| \log n)$ was given in [3], along with a randomized algorithm with the same expected running time (H is the set of hot pixels, and $|h|$ is the number of segments intersecting a pixel h). An $O((n + |I|) \log n)$ algorithm was presented in [2]. Algorithms with running times of $O(\Sigma_{h \in H} is(h) \log n)$ and $O(\Sigma_{h \in H} ed(h) \log n)$ were given in [7], where $is(h)$ is the number of segments with an intersection or endpoint in pixel h, and $ed(h)$ is the description complexity of the crossing pattern within h, which never exceeds $is(h)$. All of these algorithms rely on high precision computation.

A dynamic algorithm for snap rounding based on vertical cell decompositions was presented in [5]. A variant of the problem, iterated snap rounding, was investigated in [6, 9].

2 A modified definition of snap rounding

In this section, we investigate some properties of a snap rounded arrangement, and introduce a slight modification to the usual procedure that still produces polysegments that properly intersect only at hot pixel centers.

An ursegment s is *hot in pixel* h if h contains an endpoint of s or a proper point of intersection between s and some other ursegment. We can also say that h is *hot for* s. An ursegment s is *warm in pixel* h if s is not hot in h, yet intersects h and h is hot for at least one other ursegment s'. We can also say that h is warm for

*School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada, V5A 1S6, binay@cs.sfu.ca; research is partially supported by NSERC, MITACS, and Safe Software

†School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada, V5A 1S6, jpsember@sfu.ca

s. We can extend these definitions to deal with columns of pixels: i.e., ursegment s is hot within column c.

Note that the hot pixels for an arrangement of ursegments are exactly those pixels that are hot for some ursegment, and that ursegments that are hot or warm within a pixel are snapped to that pixel.

If ursegment s is hot in column c, let H be the set of all pixels in c that are hot for s. The uppermost and lowermost pixels in H are *external hot pixels with respect to* s.

Lemma 1 *Let h be a hot pixel which is not external with respect to any of the set of ursegments. Let s be a ursegment that is hot in h. The portion of the polysegment for s intersecting h will be a vertical line segment bisecting h.*

Now consider a new definition of snap rounding, in which a pixel is only hot if under the old definition it is an external hot pixel for some ursegment.

Theorem 2 *Using our new definition of snap rounding, polysegments of a set of ursegments will properly intersect only at hot pixel centers.*

Proof. Assume by way of contradiction that with new snap rounding, some pair of ursegments s_1 and s_2 yield polysegments that properly intersect at a point that is not a hot pixel center.

The continuous deformation introduced by Guibas and Marrimont [5] can be used to show that if such an intersection occurs, it can only occur at a pixel that was hot under the old definition but not under the new one. Let h be such a pixel. Since h is hot under old snap rounding, but not external for any ursegment, there must exist two or more ursegments that are hot in h; call this set Q. Let R be the set of any remaining ursegments that intersect h.

Let b be the vertical bisector of h. By lemma 1, the polysegments for Q contain b, so s_1 and s_2 cannot both belong to Q; and the continuous deformation can be used to prove that s_1 and s_2 cannot both belong to R. Therefore, without loss of generality we can assume $s_1 \in Q$ and $s_2 \in R$, and that the proper intersection in question must lie on b.

Let c be the column of pixels containing h. We now show that s_2 cannot be snapped to any pixel in $h' \in c$. First, $h' \neq h$, by the definition of set R and the fact that h is not external with respect to any ursegments. If h' is some other pixel in c, then due to the monotonicity of the polysegment for s_2, it cannot have a proper intersection with b, a contradiction. By the same reasoning, we see that s_2 cannot have an endpoint in c. Thus s_2 must cross c from left to right.

Lemma 1 can be used to show that every ursegment $q \in Q$ must be associated with two hot pixels in c that are external with respect to q, and that lie on opposite sides of h. Let h_1 and h_2 be such a pair of hot pixels for some q. Since these are hot pixels under the new definition as well, s_2 cannot intersect either of them; otherwise, s_2 would be snapped to them, and we know s_2 is not snapped to any pixel in c.

Since s_2 intersects h, and s_2 does not intersect h_1 or h_2, s_2 must cross the stack of pixels between h_1 and h_2 from left to right. But then s_2 must intersect q, causing s_2 to be snapped to one of these pixels within c; a contradiction. □

Theorem 3 *Using the new definition of snap rounding, a set of n ursegments produces $O(n)$ hot pixels in any one column.*

3 Algorithm One

We first present an algorithm that given a set of ursegments as input, generates the polysegments of this set. It performs a single vertical sweep of the plane by a strip of unit width, stopping only at integer coordinates.

For simplicity, we assume no ursegment is vertical or has length zero. These special cases can be dealt with easily; we omit the details for lack of space.

One of the strengths of our algorithm is that we never need to clip an ursegment to a pixel boundary, a procedure requiring high precision. We also don't need to calculate the exact intersection point of two ursegments, only the pixel containing that point (which can be calculated and represented using integers only). We will show that we can still detect and process every such intersection point when stopping the sweep strip only at column boundaries.

Actually, we need to stop at the centerline of the columns as well, since segments will start and stop at points on these lines. To allow this, we stretch each ursegment horizontally:

$$x' = 2x + 1 \tag{1}$$

A half pixel $H(x, y)$ is the set of all points $(a, b) \in \mathbb{R}^2$ such that $x \leq a < x + 1$ and $y - \frac{1}{2} \leq b < y + \frac{1}{2}$, for $x, y \in \mathbb{Z}$. In words, it is the unit square centered at $(x + \frac{1}{2}, y)$, excluding the top and right boundary edges. A full pixel is a pair of half pixels $\{H(x, y), H(x + 1, y)\}$ where $x = 2 \cdot i_{\in \mathbb{Z}}$. It follows that a full pixel is hot iff either of its component half pixels contains an ursegment intersection or endpoint. Equation (1) allows each original full pixel to be represented by two half pixels, each with integer coordinates. Note that the vertical sides of the full pixels and the ursegment endpoints lie on the vertical sides of the half pixels.

A *sweep column* at $x = x'$ is the column of half pixels $\{H(x, y) \mid x = x'\}$, and is denoted $W(x')$. A *snap column* at $x = 2 \cdot i_{\in \mathbb{Z}}$ is the pair of adjacent sweep columns $\{W(x), W(x + 1)\}$.

Our algorithm performs a modified vertical Bentley and Ottman [1] plane sweep with a strip, stopping at a sweep column if an ursegment endpoint is within the column, or if ursegments that are neighbors in the active list intersect within the column.

We must specify a total order relation to order the ursegments within the active list. Let x be the position of the sweep column. Observe that the ordering of parallel ursegments does not depend upon x, and is thus trivial to calculate. For other pairs of ursegments (s_1, s_2), we calculate h, the pixel containing their intersection point, and define the function *upper* which returns the ursegment that contains the upper of the two portions of the ursegments that lie to the left of the pixel column containing h. We then define the ordering relation $>_x$ as follows ($\oplus$ is exclusive or): $s_1 >_x s_2 \equiv (upper(s_1, s_2) = s_1) \oplus (h_x < x)$.

We augment each ursegment with three additional fields. The *heatRange* field records the highest and lowest hot pixels known to intersect the ursegment within the current snap column. We say we are *adding a pixel to a ursegment's heatRange* to mean that we are initializing or expanding the range as necessary to include the new pixel. The *heatAbove* and *heatBelow* fields are pointers to other ursegments, and their purpose is explained later.

There are three sequential processes that occur within a snap column: the *sweep* process, the *hot pixel* process, and the *snap* process.

3.1 The sweep process

When the sweep stops at a new snap column, we initialize a *snap set* of ursegments to the empty set. This set will contain ursegments that are hot within the snap column, as well as any ursegments that are ever adjacent to such ursegments within the active list.

We add each ursegment that is stopping at the sweep column's left edge to the snap set, remove it from the active list, and add its stopping endpoint to its *heatRange*.

We perform a similar procedure for each ursegment that is starting at the current sweep column's left edge: we insert it into the active list, add it to the snap set, and add its starting point to its *heatRange*.

We next query the event queue for all *seed events*: currently known intersection events occuring in the current sweep column. After pushing these events onto a stack, we pop each event, and test if it represents ursegments that are still neighbors within the active list. If not, we ignore the event (this pair will eventually become neighbors again, and will be processed then). Otherwise, we add the pixel containing their intersection point to both ursegment's *heatRange* fields, add the ursegments to the snap set, and exchange their positions within the active list. We test for intersection events that will occur between the ursegments and their new neighbors, and if the intersection point will occur within the current sweep column, we add the event to the stack. If the stack is not yet empty, we pop another event and repeat. Once all stacked intersections have been processed, the active list will be in the correct order for the right side of the current sweep column.

Whenever we process a hot pixel h for a segment s' with an active list neighbor s, we take some additional actions that will aid us in finding warm ursegments later. First, if $s.heatRange$ is defined, we do nothing, since s is known to be hot. Otherwise, if s' is below s, then if $s.heatBelow$ is undefined or has a *heatRange* that ends below that of s', we make $s.heatBelow$ point to s'. We perform a similar action if s' is above s and $s.heatAbove$ has a *heatRange* that starts above that of s'.

If the next intersection or endpoint event to be reported will occur within the same snap column, we repeat the sweep process (retaining the contents of the snap set). Otherwise, we continue with the hot pixel process.

3.2 The hot pixel process

Every ursegment in the snap set with a defined *heatRange* field will be exactly those ursegments that are hot within the column. We construct a linked list of unique hot pixels, sorted by y, from these *heatRange* fields. For quick access during the snap process, we replace each *heatRange* field's hot pixels with their corresponding pointers into this list.

Every hot ursegment may have exchanged positions within the active list, so we have the tree predict intersection events for these ursegments, in preparation for the next sweep process.

3.3 The snap process

We first process every hot ursegment by examining every ursegment s in the snap set with a defined *heatRange* field. As a result of the previous process, s will have at least one pointer to a pixel within the hot pixel list that it should be snapped to. We can iterate above and below this pixel to find all pixels that are warm or hot for s, snapping s to each (and expanding $s.heatRange$) as we go.

Next, we process every potentially warm ursegment by examining those ursegments s with undefined *heatRange* fields. The *heatAbove* and *heatBelow* fields, if defined, will point to the segment s' with the nearest hot pixel to either side of s. If s intersects $s'.heatRange$, then one of the two hot pixels in $s'.heatRange$ will be warm for s and can be used as a starting point to iterate through all the pixels that are warm for s. We snap s to these pixels, add the pixels to $s.heatRange$,

and recursively repeat this procedure with each (cold) neighbor of s, using s as the 'heat source'.

Once a snap column is processed, we reinitialize any ursegment fields that were modified, in preparation for processing another column.

4 Performance

Our algorithm uses a variant of a B+ tree to store the active list of ursegments that intersect the sweep column, ordered by the position of their intersections along the sweep column. The tree supports $O(\log n)$ insertion, deletion, and search operations, and $O(1)$ traversal through the active list. The intersection events are also stored within this tree. We use a separate tree to store the ursegment endpoint events.

The seed events reported by the tree during the sweep process each require at most $O(\log n)$ time. Note that an ursegment can occur in at most two seed events, so the cost of extracting all such events for a sweep column c is $O(is(c)\log n)$, where $is(c)$ is the number of ursegments that are hot within c. Note also that the $O(\log n)$ cost of processing each endpoint event can be included in this term.

We can exchange ursegments within the active list in $O(1)$ time if we associate each ursegment with a pointer and access the ursegments via these pointers; we omit the details for brevity.

The number of exchanges that occur during all sweep processes is bounded by $|I|$, so the running time for the sweep processes aggregated over every sweep column is $O(|I| + \Sigma_c\ is(c)\log n)$.

The hot pixel process sorts the hot pixels; by theorem 3, there are $O(is(c))$ such hot pixels, so this cost can be included in that of the sweep process. For each of the $is(c)$ intersecting segments, the tree spends $O(\log n)$ recalculating intersection events (details are omitted for lack of space), so this cost can also be included in the sweep process.

The snap process spends $O(1)$ time generating each polysegment fragment. Since I^*_m is the set of all such fragments, the running time of the complete algorithm is $O(|I| + \Sigma_c\ is(c)\log n + |I^*_m|)$.

5 Algorithm Two

As observed in [6], $|I^*_m|$ can be as much as $\Theta(n^3)$. Our second algorithm uses a similar approach to that of [2] to generate I^*, the rounded arrangement of the ursegments instead of their individual polysegments.

We perform three plane sweeps. The first is a vertical sweep to find the set of hot pixels, and is simply the first algorithm with the snap process omitted.

We partition the ursegments into two sets: those with slopes m between -1 and 1, and those whose slopes are outside of this range. The second sweep is a vertical sweep that includes all hot pixels and the ursegments from the first set, while the third sweep is a horizontal sweep which includes the hot pixels and the ursegments from the second set.

We organize the ursegments into bundles, generate an arc between hot pixels where a bundle intersects the hot pixels, and split the bundles as necessary at hot pixels. Using bundles in this way imposes a logarithmic cost on each arc generated, which results in a running time of $O(|I| + \Sigma_c\ is(c)\log n + |I^*|\log n)$.

6 Conclusion

We have presented two algorithms to perform snap rounding. Both use simple integer arithmetic, are robust, are practical to implement, and are easily adapted for segments whose endpoints are rational numbers. They improve upon existing algorithms, since existing running times either include a logarithmic factor in $|I|$, or depend upon the number of segments interacting within a particular hot pixel, whereas ours are linear in $|I|$ and depend upon the number of segments interacting in an entire hot column, a much coarser partition of the plane.

An applet demonstrating both algorithms is available at `http://www.sfu.ca/~jpsember/snap.html`.

References

[1] J. L. Bentley and T. Ottman. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[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.

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

[4] D.H. Greene. Integer line segment intersection. Unpublished manuscript.

[5] L. J. Guibas and D. H. Marimont. Rounding arrangements dynamically. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 190–199, 1995.

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

[7] J. Hershberger. Improved output-sensitive snap rounding. In *Proc. 22nd Annu. ACM Sympos. Comput. Geom.*, 357–366, 2006.

[8] J. D. Hobby. Practical segment intersection with finite precision output. *Comp. Geom.: Theory and Appl.*, 13:199–214, 1999.

[9] E. Packer. Iterated snap rounding with bounded drift. In *Proc. 22nd Annu. ACM Sympos. Comput. Geom.*, 367–376, 2006.