Efficient Snap Rounding in Square and Hexagonal Grids using Integer Arithmetic

Boaz Ben-Moshe[*] Binay K. Bhattacharya[†] Jeff Sember[‡]

February 12, 2009

Abstract

In this paper we present two efficient algorithms for snap rounding a set of segments to both square and hexagonal grids. The first algorithm takes n line segments as input and generates the set of snapped segments in $O\big((n+k)\log n+|I|+|I^*_m|\big)$ time, where k is never more than the number of hot pixels (and may be substantially less), $|I|$ is the complexity of the unrounded arrangement I, and I^*_m is the multiset of snapped segment fragments. The second algorithm generates the rounded arrangement of segments in $O(|I|+(|I^*|+\Sigma_c\, is(c))\log n)$ time, where $|I^*|$ is the complexity of the rounded arrangement I^* and $is(c)$ is the number of segments that have an intersection or endpoint in pixel row or column c. 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 an $|I|\log n$ term, or depend upon the number of segments interacting within a particular hot pixel h ($is(h)$ and $ed(h)$ [9], or $|h|$ [4]), whereas ours depend on $|I|$ without the $\log n$ factor and are either independent of the number of segments interacting within a hot pixel (algorithm 1) or depend upon the number of segments interacting in an entire hot row or column ($is(c)$), which is a much coarser partition of the plane (algorithm 2).

1 Introduction

Many geometric algorithms are difficult to implement in practice due to robustness and precision problems. This is because the algorithms often assume the availability of an infinite-precision real arithmetic machine (or RAM [14]). One approach that has been taken to address this problem is the development of

[*]Dept. of Computer Science, Ariel University Center, Ariel, 40700, Israel, benmo@ariel.ac.il

[†]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

[‡]UBC Computer Science, Vancouver, B.C., Canada, V6T 1Z4, jpsember@cs.ubc.ca; research is supported by NSERC

algorithms that are practical and provide useful results despite using limited precision arithmetic [12, 15, 16]. The *snap rounding* algorithms presented here fall into this category.

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 the plane. We wish to round the arrangement I of S to a grid of pixels. In this note, we assume that segment endpoints are integers, 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. Following Guibas and Marimont [7], 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*.

We define the *fundamental property* of a snap rounded arrangement as this: fragments that are not identical will intersect only at a common endpoint.

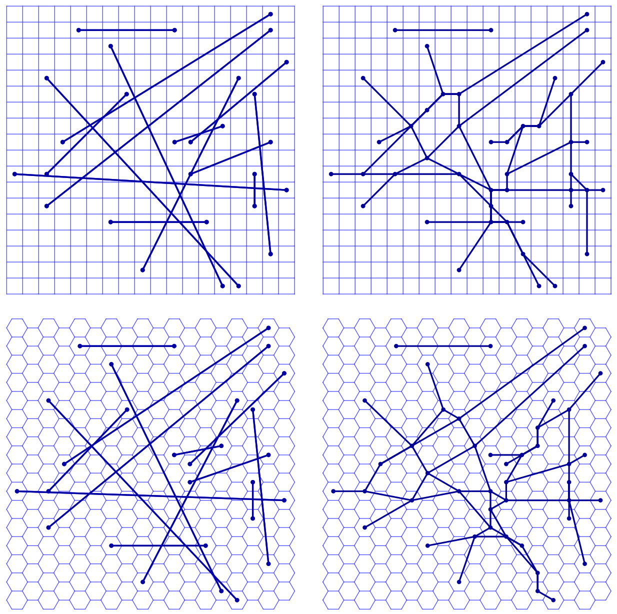
Figure 1: Snap rounding, square and hexagonal grids.

Snap rounding was introduced independently by Greene [5] and Hobby [10]. An algorithm with running time $O(n \log n + \Sigma_{h \in H} |h| \log n)$ was given in [4], 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 pixel h). An $O((n + |I|) \log n)$ algorithm was presented in [3]. 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 [9], 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 $is(h)$. We here note that all of these algorithms require floating point computation to order some intersection events within a pixel.

A dynamic algorithm for snap rounding based on vertical cell decompositions was presented in [7]. A variant of the problem, iterated snap rounding, was investigated in [8, 13].

In the next section, we describe how snap rounding can be applied to different grid types, and provide a slightly modified definition of snap rounding that improves the efficiency of our algorithms. We present our first algorithm in section 3, and analyze its efficiency in section 4. We present our second algorithm in section 5. We then conclude with some remarks and areas of future research.

2 Preliminaries

In this section, we describe the grid types our algorithm can work with, and introduce a slight modification to the snap rounding procedure that improves the algorithm's efficiency while maintaining the fundamental property of snap rounding.

There are three possible regular tilings of the plane: square, hexagonal, or triangular [6]. We examine each of these possibilities in turn.

We define a *center pixel* to be one whose coordinates are mapped to the point in the pixel's center, and a *corner pixel* to be one whose coordinates are mapped to a vertex on the pixel's boundary.

It will simplify our algorithm if pixel centers and vertices have integer coordinates. To achieve this, we transform each ursegment from its original *grid space* of center pixels to *strip space*, a partition of the plane into corner pixels. From this point on, *pixel* refers to a grid space pixel, while *spixel* refers to a strip space pixel. Each spixel is square, and is a corner pixel with its coordinates mapped to its bottom left vertex. Thus a point $(a, b) \in \mathbb{R}^2$ lies within the spixel with coordinates $(\lfloor a \rfloor, \lfloor b \rfloor)$. Observe that each spixel is closed at its left and bottom edges, and at its bottom left vertex. A column of spixels forms a *strip*. As we shall see, more than one strip may intersect a particular column of pixels, and (in the case of hexagonal grids) a particular strip can intersect more than one column of pixels.

2.1 Square grids

Our square grid partitions the plane into square center pixels. The strip transformation for square pixels is

$$\begin{aligned} x' &= 2x \\ y' &= 2y\ . \end{aligned}$$

In strip space, each square pixel appears as in figure 2. As with spixels, each square pixel is closed at its bottom and left edges, and at its bottom left vertex.

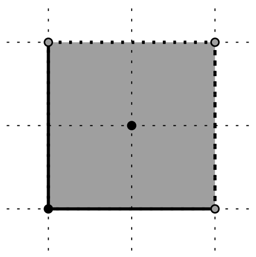
Figure 2: Square pixel in strip space

2.2 Hexagonal grids

In recent years, hexagonal grids have received much attention from the computer graphics and machine vision communities [11]. It is therefore useful to consider if snap rounding can be applied to these grids.

Before discussing hexagonal grids in detail, we should first prove that snap rounding is required on such grids, and that it is possible with such grids. Simple examples showing that it is required for square grids are easy to find; see, e.g., [10], figure 1. A similar example for hexagonal grids is shown in figure 3, involving three ursegments. Observe that after snapping the intersection point of two ursegments to its containing pixel's center, a spurious intersection is created between snapped fragments.

We now prove that snap rounding is possible with hexagonal grids.

Theorem 1 *The fundamental property of snap rounding is preserved when snap rounding is performed on a hexagonal grid.*

Proof. Guibas and Marimont [7] presented a continuous deformation concept to prove that the fundamental property holds for square grids. In their deformation, each ursegment is split into a number of subsegments by inserting a breakpoint (a node) wherever the ursegment crosses a hot pixel boundary. Each hot pixel is then linearly scaled down in the x-dimension, dragging any intersecting nodes along, until it has become a hot 'stick'. A similar scaling operation is then performed in the y-dimension, reducing the hot stick to a single point at the original pixel's center. In order for the fundamental property to be violated, at some point during the deformation two subsegments must intersect (at a point that is not a common endpoint), which implies that a subsegment that is external to the hot pixels must enter a deforming hot pixel. In order for this to occur, it must overtake a vertex of the hot pixel. Note that as a subsegment (or pixel edge) is deformed during the x (resp., y) scaling operation, each point on the subsegment follows a path that is parallel to the x (resp., y) axis. Thus for a subsegment to overtake a pixel vertex v, some point p on the subsegment must be moving in the same direction as, and faster than, v. But

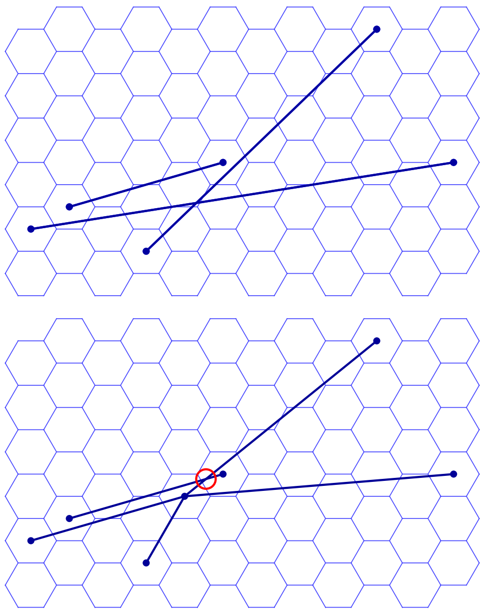

Figure 3: Snap rounding is required on hexagonal grids

p's velocity is a convex combination of the velocities of its endpoints, and every vertex is moving at least as fast as any endpoint; hence p can never overtake v.

To prove that the fundamental property holds with hexagonal grids, we use a similar deformation as follows. First, we scale the grid and ursegments in y so the interior angles of the leftmost and rightmost vertices of the hexagonal pixels are $90°$. Next, we move each point on a pixel boundary (including nodes) at constant speed towards the vertical line through the center of each pixel, stopping each point when it reaches this line. Observe that four of the six hexagon vertices will reach this line simultaneously. When this occurs, we stop the deformation. Note that during the deformation, every point on a subsegment or pixel boundary will have moved on a path parallel to the x-axis. Thus the previous argument can be applied to show that no subsegment can enter a hot pixel during this first stage of the deformation.

When the first stage is complete, each hot pixel has contracted to a square (rotated $45°$ from the axes). We move the axes so the pixels are axis aligned, and apply the original two-step deformation of Guibas and Marimont to reduce the hot pixel to the original hexagon's center point. Their argument can be directly applied to prove that no ursegment ever enters the hot pixel during these final deformation steps. $\square$

For our hexagonal grid[1] (figure 4), we adopt a coordinate system that assigns integer coordinates to each hexagon. In strip space, each hexagonal pixel

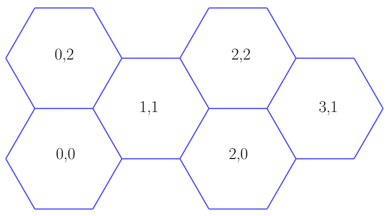

Figure 4: Hexagonal grid

appears as in figure 5. Note that each hexagon is closed at three edges and two vertices. The strip transformation for these pixels is

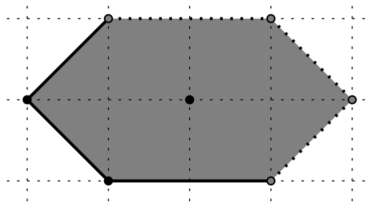

Figure 5: Hexagonal pixel in strip space

$$x' = 3x$$
$$y' = y \ .$$

2.3 Triangular grids

Snap rounding is also required on triangular grids, as figure 6 shows. Unfortunately, the fundamental property of snap rounding does not apply with these grids; see figure 7. The hot pixels are highlighted. As ursegment a is snapped to become polysegment a', it is not snapped to hot pixel j, since a does not intersect j; but after a is snapped to i, a' passes below the center of j, causing a problem intersection.

2.4 A modified definition of snap rounding

We now introduce a minor modification to the snap rounding procedure that improves our algorithm's efficiency while maintaining the fundamental property of snap rounding. We begin with some terminology.

[1] If the desired hexagonal grid has pixels that are rotated 90 degrees to our grid, then rotating the input ursegments and output fragments will allow our algorithm to use the grid of figure 4.

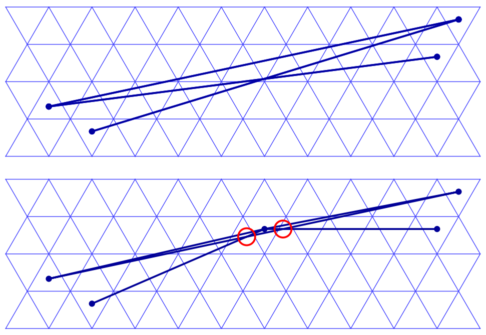
Figure 6: Snap rounding is required on triangular grids

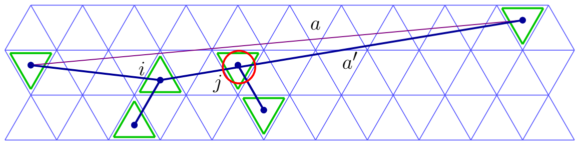

Figure 7: Snap rounding fails with triangular grids

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. An ursegment s is *warm in* h if s intersects h, h is a hot pixel, yet s is not hot in h. An ursegment s is *heated in* h if it is hot or warm in h. 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 some ursegment is hot within, and that ursegments that are hot or warm within a pixel are snapped to that pixel.

Consider the column of 'partial' pixels that result from the intersection of a pixel column c and a strip p. Let c' be the subset of these partial pixels that contain an endpoint of an ursegment s, or an intersection point between s and another ursegment. The (full) pixels containing the uppermost and lowermost partial pixels in c' are *extremal heat pixels of* s *with respect to* c *and* p.

In our new definition of snap rounding, a pixel is only hot if under the old definition it is an extremal heat pixel of some ursegment.

Theorem 2 *The fundamental property of snap rounding is preserved under the new definition of snap rounding.*

Proof. Assume by way of contradiction that with new snap rounding, some pair of ursegments s_1 and s_2 yield nonidentical fragments that intersect at a point that is not a common endpoint.

The continuous deformation of [7] can be used to show that if such an intersection occurs, it can only occur within a pixel that was hot under the old definition but not under the new one. Let h be such a pixel, and c (resp. p)

the pixel column (resp. strip) containing the problem intersection. Since h is hot under old snap rounding, but not extremal for any ursegment, $h \cap p$ cannot contain any ursegment endpoints, and there must exist two or more ursegments that have an intersection in $h \cap p$; call this set Q.

Observe that for every ursegment $q \in Q$, there exist within c extremal heat pixels h^+ above h and h^- below h. For any pair of ursegments q_1 and q_2 intersecting in h, both q_1 and q_2 will be snapped to the lower of (h_1^+, h_2^+) (denoted h_{12}^+), the higher of (h_1^-, h_2^-) (denoted h_{12}^-), and to any extremal heat pixels between these (which in the hexagonal case may include any extremal heat pixels in an adjacent column intersecting p). This implies that the section of the polysegments for q_1 and q_2 between h_{12}^+ and h_{12}^- must be identical, which includes the portions intersecting $h \cap p$. Hence s_1 and s_2 cannot both belong to Q, so s_1 cannot intersect s_2 prior to snapping. The continuous deformation can then be used to show that the polysegments from s_1 and s_2 cannot intersect in h, a contradicition. □

The value of theorem 2 lies in the fact that we can ignore the non-extremal heat pixels for each ursegment within a particular pixel column / strip pair.

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 horizontal sweep of the plane by a vertical *sweep strip* of unit width, which stops only at integer coordinates. For the moment, we assume no ursegment is vertical or has length zero. We will discuss how to include such ursegments later.

We will be interested in the sign of the slope of an ursegment. For ease of exposition, we treat ursegments with zero slopes as if their slopes are positive.

One of the strengths of our algorithm is that we never need to clip an ursegment to a pixel boundary, a procedure requiring floating point computations. 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 despite stopping the sweep strip only at integer coordinates.

Our algorithm performs a modified Bentley and Ottman [1] plane sweep with a vertical strip, stopping at a strip boundary if an ursegment's left endpoint lies on the left boundary of the strip, if an ursegment's right endpoint lies on the right boundary of the strip, or if ursegments that are neighbors in the active list intersect within the strip.

We must specify a total order relation to order the ursegments within the active list. Let x be the position of the sweep strip. 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 $h = (h_x, h_y)$, the spixel containing the intersection point of their containing lines, 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 strip containing h. We then define the ordering relation $>_x$ as follows ($\oplus$ is exclusive or): $s_1 >_x s_2 \equiv (upper(s_1, s_2) = s_1) \oplus (h_x < x)$.

3.1 Data structures

Our algorithm uses a tree to store both the active list of ursegments intersecting the sweep strip, and pending intersection events between neighboring ursegments. The active list is sorted according to the total order relation described previously. Intersection events are sorted lexicographically according to the spixel containing the intersection.

We maintain two separate priority queues for the ursegments: one for starting endpoints, and one for stopping endpoints. These queues sort the points lexicographically in the same manner as the active list.

A *HeatList* is a linked list of ursegments that are heated within a particular sweep strip (or, in some cases, that are immediate neighbors of such ursegments). We maintain a *HeatListQueue* for a set of contiguous sweep strips, where each element in the queue contains two HeatLists: one with the ursegments ordered according to their position as they enter the strip, and the other ordered according to their position as they leave the strip.

For each ursegment, we maintain a queue of *SegHeatInfo* records. These records, which are associated with a particular strip, record the highest and lowest known heat pixels for the ursegment within each pixel column intersecting the strip. Each queue needs to contain only as many records as there are strips intersecting a pixel column (2 for square grids, 4 for hexagonal grids), and since at most two pixel columns can intersect any one strip (with hexagonal grids, for instance), the maximum size of this queue is bounded by a small constant.

When we record a heat pixel within an ursegment's SegHeatInfo queue, we will say that we are *registering* the pixel with the ursegment.

There are three main processes that occur within the algorithm: the *sweep* process, the *hot pixel* process, and the *snap* process.

3.2 The sweep process

For each sweep strip, a HeatList is constructed for the ursegments that are beginning at the left edge of the strip, by popping ursegments from the (starting) endpoint queue. Each such ursegment is added to both the tree and the HeatList, and its starting endpoint is registered.

A similar procedure is performed for each ursegment that is stopping at the right edge of the strip. A HeatList is constructed of these ursegments, and each stopping endpoint is registered with its ursegment. This HeatList also includes the immediate neighbors of a stopping ursegment, since by the time this HeatList is examined, the ursegment will have been removed from the tree and its original neighbors will not be known.

Next, the tree is queried for all *seed events*: intersection events known to occur within the current sweep strip. All seed events are between adjacent ursegments. These events are pushed onto a stack, and a HeatList is constructed of the ursegments involved.

There are now three HeatLists: starting ursegments, stopping ursegments, and intersecting ursegments. These are merged into a single 'left side' HeatList, which includes all ursegments known to be hot within the sweep strip. A 'right side' HeatList is constructed, which is initially an exact copy of the left side HeatList, but will be manipulated so the order of its ursegments corresponds to how the ursegments leave the strip on the right side. The stack of seed events generated earlier is now processed.

Each event is popped, and tested to see if it represents ursegments that are still neighbors within the active list. If not, the event is ignored. Otherwise, the pixel containing their intersection point is registered with both of the ursegments, and their positions within the active list are exchanged. Their positions within the right side HeatList are also exchanged, and new neighbors of these ursegments are inserted into this HeatList as required (so that if they are involved in intersections, they will also be present in the HeatList). Tests for intersection events that will occur between the ursegments and their new neighbors are performed, and if the intersection point will occur within the current sweep strip, the event is added to the stack. If the stack is not yet empty, another event is popped and processed. Once all stacked intersections have been processed, the active list will be in the correct order for the right side of the current sweep strip, as will the right side HeatList. The left side and right side HeatLists are now added to the HeatListQueue.

The next position of the sweep strip is determined, as well as which pixel columns intersect the current sweep strip but not the next one. For each of these pixel columns, the hot pixel and snap processes (described in the following sections) are performed.

The sweep strip is advanced by one, so the active list (which has been manipulating directly) now agrees with the rest of the tree. At this point an iteration through the stopping ursegments HeatList is performed to remove any ursegments that terminated in the previous sweep strip.

If necessary, the sweep strip is advanced and the sweep process is repeated.

3.3 The hot pixel process

This process constructs a set of the hot pixels within a particular pixel column, sorted by y-coordinate.

Lemma 3 *If S is a set of ursegments crossing a strip, each ursegment is associated with the range of pixels that it is hot within, and S_1 (resp. S_2) are the ursegments of S ordered by their intersection points with the left (resp. right) boundary of the strip, then the ordered extremal heat pixels for S can be constructed in $O(|S|)$ time.*

Proof. For the moment, assume we are dealing with square grids. First, S_1 and S_2 are partitioned by the sign of each ursegment's slope into ordered sets

S_1^+, S_1^-, S_2^+, and S_2^-. Let $s^\downarrow$ and $s^\uparrow$ represent the y-coordinates of the lower and upper hot pixels associated with ursegment s. Observe that if $s_a \prec s_b$ in S_1^+, then the lowest extremal heat pixel for s_a cannot be higher than any hot pixel intersected by s_b. A backward iteration through S_1^+ is performed, examining each adjacent pair $s_a \prec s_b$, replacing $s_a^\downarrow$ with $\min(s_a^\downarrow, s_b^\downarrow)$. This is followed by a forward iteration through S_1^-, and for each adjacent pair $s_a \prec s_b$, $s_b^\uparrow$ is replaced with $\max(s_a^\uparrow, s_b^\uparrow)$. Similar backward (resp., forward) iterations through S_2^- (resp., S_2^+) are performed, to find the extremal heat pixels for the negatively-sloped ursegments in S.

The lower extremal heat pixels of S_1^+ and S_2^- are now in monotonically increasing order, as are the upper extremal heat pixels for S_1^- and S_2^+ (note that since every ursegment $s \in S$ crosses the strip, both its lower and upper extremal heat pixels can be found in one of these four sequences). These four sequences can be merged in linear time to yield the extremal heat pixels for the ursegments of S.

One complication exists with hexagonal grids. Observe that the boundary of a pixel column does not coincide with any sweep strip boundary, so the ordering of the ursegments within S_1 and S_2 may not be the same as the desired ordering, which is by the ursegments' intersections with the pixel column boundary. In figure 8, ursegment $a \prec b$ with respect to the (left) boundary of the pixel column, yet $b \prec a$ in S_1. To handle this inconsistency, the ordering of the

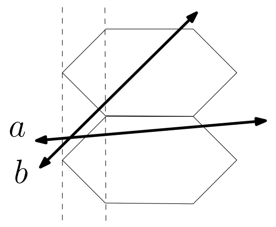

Figure 8: Hexagonal pixel column and ursegment order

ursegments S_1 (or, alternatively, S_2) is corrected before applying the above procedure. The differences between the two orderings are localized to the interfaces between adjacent pixels, so this correction can be performed in $O(|S|)$ time. The procedure is straightforward; we omit the details. $\square$

To generate the sorted list of hot pixels for an entire pixel column, this procedure is applied for each strip intersecting the pixel column. From the HeatListQueue, the left and right side HeatLists are gathered for every strip intersecting the pixel column. Lemma 3 can be applied to the HeatLists, since (i) they contain ursegments ordered by their intersections with the strip boundaries, and (ii) the sweep process will have registered every hot pixel with its associated ursegment(s). This procedure is applied to the (heated) ursegments within these HeatLists, and the resulting hot pixel lists for each strip are merged into a list for the entire pixel column.

3.4 The snap process

The sorted set of hot pixels within a pixel column are now used to generate fragments for any ursegments intersecting hot pixels within the column. From the HeatListQueue, the left and right side HeatLists are gathered for every strip intersecting the pixel column. For each HeatList, iterations through the positively-sloped ursegments and the hot pixel list are performed, detecting a hot pixel that intersects the ursegment. If one is found, the ursegment is snapped to every hot pixel intersecting the ursegment, and the neighbors of the ursegment and the hot pixel are processed recursively, to see if these ursegments are warm within the hot pixel as well.

This procedure is repeated with negatively-sloped ursegments. The direction of iteration through the ursegment and hot pixel sets depends upon which side of the strip the HeatList is associated with and the slopes of the ursegments, in a manner similar to that of the procedure from the proof of Lemma 3. Once an ursegment is snapped to a pixel within the column, it is ignored it if it appears again (in a HeatList, or as a neighbor to an ursegment within the recursion).

The snap points must be generated in the correct order for an ursegment. For square grids, this is simply a matter of iterating through the hot pixel list in the correct direction. For hexagonal grids, the procedure is slightly more complicated, as several hot pixels from consecutive pixel columns might be interleaved within a correct snap order. Observe that this interleaving is localized to at most two adjacent pixel columns, so a postprocessing step can fix the snap ordering for an ursegment in time linear in its number of snap points.

The discrepancy between the order of ursegments in HeatLists and their order of intersection with a hexagonal pixel columns (described in section 3.3) introduces another complication here. Since the discrepancy is localized to the interface of two adjacent pixels, this is easily dealt with by using a window of size two when iterating through the hot pixel list during the snap process; we omit the details.

3.5 Vertical ursegments

Vertical ursegments (and zero length segments, which can be treated as vertical ursegments) present a complication for our algorithm, since they are difficult to incorporate into the total order relation described earlier, and they do not span the width of a sweep strip. We now describe how they can be treated as a special case.

We avoid problems with sorting the vertical ursegments by constructing *bracket* ursegments for each vertical ursegment's endpoints. Each is a horizontal ursegment extending to the right from the endpoint, and pose no difficulties for sorting purposes.

Only the start endpoints for vertical ursegments are added to the endpoint queue. In the sweep process, any vertical ursegment start endpoints that are popped from the queue are used to construct a vertical ursegment HeatList, which consists of any ursegments within the active list that are intersected by the vertical ursegment, or that would be immediate neighbors of its associated bracket ursegments (each bracket can be inserted into the tree temporarily to determine these neighbors).

The hot pixels for intersections between vertical and active ursegments are explicitly generated. Any such pixel is registered with the appropriate active ursegment, and the vertical ursegment endpoints are registered with both the vertical ursegments and their bracket counterparts. Since the active list is already sorted, by presorting the vertical ursegment endpoints, this procedure can be done in time linear in the number of fragments generated, so it does not affect the running time of the algorithm.

The vertical ursegment HeatList is merged together with the starting ursegment, stopping ursegment, and intersecting ursegment HeatLists, as described in section 3.2.

After the intersection events have been processed to generate the right side HeatList, the bracket ursegments for the vertical ursegments are merged into the left HeatList. (There is no point in doing this before the right HeatList is constructed, as the vertical ursegments do not extend to the right side of the strip.)

In the snap process, when a bracket ursegment is encountered, its associated vertical ursegment is processed instead.

Using the above steps, vertical (and zero length) ursegments can be included in our algorithm's input, without affecting its asymptotic running time.

4 Performance

Processing a starting and stopping endpoint for an ursegment requires $O(\log n)$ time. The total time spent processing ursegment endpoints is thus $O(n \log n)$.

Our algorithm uses a B+ tree [2] to store the active list of ursegments, and supports $O(\log n)$ insertion, deletion, and search operations. The tree consists of internal and leaf nodes, and each ursegment in the tree maintains a pointer to its location within a leaf node. Leaf nodes are joined in a doubly-linked list, so traversal from an ursegment to its neighbor in the tree can be performed in $O(1)$ time.

We augment the B+ tree to allow it to report seed events, and to allow us to efficiently process these and other intersection events during the sweep process. Each tree node contains these additional fields:

- *firstIsectEvt*, the first intersection event to occur between adjacent ursegments within this node's subtree
- *firstSeg*, the first ursegment in the node's subtree
- *valid*, true iff *firstIsectEvt* and *firstSeg* have been calculated for this node
- *invalidated*, true if this node has been marked as invalid

- *invalidatedAt*, the sweep strip position when the node was last marked as invalid

During the sweep process, the tree reports the seed events occurring within the current sweep strip. If a node's *firstIsectEvt* is null, or occurs in a strip to the right of the current strip, then the subtree at that node can be ignored. Otherwise, the subtree must be searched for seed events. Seed events between ursegments are generated within a child node (by processing the child nodes recursively) and between the last ursegment in one child subtree and the first ursegment in the next (the *firstSeg* field can be used to determine these ursegments efficiently), taking care to generate these events in the order corresponding to the ursegment's order within the active list.

While reporting a seed event may require as much as $O(\log n)$ time, it happens that all seed events that occur within a particular hot pixel h can be reported in $O(\log n)$ time plus an additional cost that can be charged to the snap process. Consider the lowest and highest ursegments involved in seed events within h. Between these ursegments within the active list lie q ursegments, each of which must intersect h. It can be shown that the cost of reporting all seed events for h is $O(2 \cdot \log n + 2q) = O(\log n) + O(q)$, and since each of the q ursegments will need to be snapped to h, the $O(q)$ term can be charged to the snap process.

The cost for reporting all seed events within a particular hot pixel (and sweep strip) is therefore $O(\log n)$. Since a pixel intersects up to four strips, the cost for reporting all seed events within a particular hot pixel is $O(4 \cdot \log n) = O(\log n)$. Reporting all seed events is thus $O(k \log n)$, where k is never greater than the number of hot pixels. We stress that this is a worst-case bound, and in many situations k will be much less. For example, if many seed events occur within a strip, but there are not many ursegments separating those involved in the seed events, then the query time for each event will be much less than $\log n$, resulting in a lower effective value of k. We have found that k is typically between $|H|/3$ and $|H|/2$, where $|H|$ is the number of hot pixels.

Constructing the HeatLists during the sweep process requires $O(1)$ time for each ursegment involved. A HeatList can contain ursegments involved in endpoint events or intersection events, or immediate neighbors of these ursegments. Thus constructing the HeatLists can be charged to the endpoint event and intersection event processing.

The number of intersections processed during all sweep processes is bounded by $|I|$. Each intersection requires exchanging ursegments within the active list, and within the right side HeatList. Ursegments can be exchanged within the active list in $O(1)$ time if each ursegment is associated with a pointer and the ursegments are accessed via these pointers. Ursegments can similarly be exchanged within the HeatList in $O(1)$ time by maintaining a pointer from an ursegment to its location within the right side HeatList. The total time spent processing these ursegment exchanges during all sweep processes is therefore $O(|I|)$.

Whenever a node within the tree is modified, i.e. during an ursegment insertion or deletion operation, the node is marked as invalid, so the *firstIsectEvent* and *firstSeg* fields are recalculated when they are next needed to find seed events. When the positions of two ursegments within the tree are exchanged during the sweep process, the appropriate nodes must be explicitly marked as invalid, since the tree is not being used to perform this operation. For each of the two ursegments, its path from leaf to root is marked as invalid. If, during this invalidation operation, a node is encountered which has already been invalidated within the current sweep strip (by checking the *invalidated* and *invalidatedAt* fields), the traversal can stop, since the remaining path to the root has already been invalidated. The cost of invalidating these paths can therefore be absorbed by the cost of processing seed events.

The total running time of all sweep processes is thus $O((n + k)\log n + |I|)$.

The hot pixel process generates a sorted list of hot pixels, but operates in time linear in the size of the HeatLists. The cost of this process can therefore be included in the cost of 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 $O\big((n + k)\log n + |I| + |I_m^*|\big)$.

5 Algorithm Two

As observed in [8], $|I_m^*|$ can be as much as $\Theta(n^3)$. Our second algorithm uses a similar approach to that of [3] to generate I^*, the rounded arrangement of the ursegments instead of their individual polysegments.

As with the algorithm of [3], we organize ursegments into *bundles*, maximal sets of ursegments that intersect the same *source* and *destination* hot pixels, without intersecting any hot pixels in between. Observe that each bundle is associated with a distinct edge of the rounded arrangement. Our bundles are simpler than that of [3], since each bundle b contains only three items: the source hot pixel *b.source*, and the boundary ursegments within the bundle, *b.lower* and *b.upper*. Since no pair of ursegments within a bundle will intersect in the space between the bundle's source and destination pixels, *b.lower* and *b.upper* are chosen to reflect the ursegment ordering within this space. We store the bundles within a *bundle tree* (i.e., a red-black tree), sorted by position along the sweep strip.

We use the same tree as in the first algorithm to store the active ursegments, and the same priority queues for the ursegment start and endpoints.

We perform three plane sweeps. The first is a plane sweep to find the set of hot pixels, and is simply the first algorithm with the snap process omitted. These hot pixels form the vertices of the rounded arrangement. Point ursegments are omitted from the subsequent sweeps, since they cannot further affect the rounded arrangement once the hot pixels are known. From this point on, then, we assume none of the ursegments are points.

The second and third sweeps generate the edges of the rounded arrangement. We now investigate how this can be done efficiently. For the moment, assume we are dealing with a square grid. Suppose that the slope m of ursegment s satisfies $0 \leq m \leq 1$,[2] and that s intersects hot pixels h_i and h_j and no hot pixels between these two, thus causing an edge to appear between h_i and h_j in the rounded arrangement. Without loss of generality, there are two types of edges between h_i and h_j to consider: (i) those where h_i is in a pixel column to the left of that of h_j, or (ii) those where h_i the lower neighbor of h_j within the same pixel column. As we shall see, the bundles that we will manipulate during these sweeps can be used to generate edges of type (i). Type (ii) edges require special treatment, and we will require the following lemma[3].

Lemma 4 *If an ursegment s with slope $0 \leq m \leq 1$ intersects hot pixels h_i and h_j (with h_i below h_j) within the same pixel column, then s belongs to some bundle b whose destination hot pixel is h_i, and either (i) $s \in \{b.lower, b.upper\}$, (ii) either $b.upper$ or $b.lower$ intersects both h_i and h_j as well, or (iii) s intersects either $b.upper$ or $b.lower$ within h_i.*

Proof. Let s be an ursegment with slope $0 \leq m \leq 1$ that intersects h_i and h_j within a pixel column, where h_i is below h_j. We first show that h_i is the destination for some bundle b containing s. Observe that since s intersects both h_i and h_j, and $m \leq 1$, s must intersect h_i before any other hot pixels within the column; and its starting endpoint cannot be in h_i, or else it cannot intersect h_j. Thus s must belong to some bundle whose destination hot pixel is h_i.

Now suppose $s \notin \{b.lower, b.upper\}$, and neither *b.upper* nor *b.lower* intersects both h_i and h_j. Now, *b.upper* either crosses h_i from left to right without intersecting h_j, or *b.upper* has an endpoint at the center of h_i. In the first case, s must remain below *b.upper* within h_i, so it cannot intersect h_b, a contradiction. In the second case, s cannot intersect h_j without also intersecting *b.upper* (since $m \leq 1$), also a contradiction. □

Lemma 4 suggests an approach to generate edges of type (ii). We associate each ursegment s with $s.hotPixel$, a pointer to the destination hot pixel for the most recent bundle known to contain s. We can set $s.hotPixel$ for the boundary ursegments for each bundle, and propagate these pointers to other ursegments within the bundle only when ursegment intersections are processed. Whenever we set this pointer for an ursegment, we test whether an edge of type (ii) needs to be generated. The details are explained in the following sections.

The second sweep is a horizontal sweep with a vertical strip, and consists of three processes: the *split process*, the *sweep process*, and the *create process*, described below. Prior to the sweep, we initialize the ursegment and bundle trees, and populate the endpoint priority queues. For square grids, we include only those ursegments with slopes m satisfying $-1 \leq m < 1$; for hexagonal grids, the slopes must satisfy $-\frac{1}{2} \leq m < \frac{\sqrt{3}}{2}$. The first sweep has generated the hot pixels already sorted lexicographically by column then row, so no sorting of the hot pixels is required.

[2]The following discussion can easily be modified if $-1 \leq m < 0$.

[3]Lemma 4 can be easily adapted to hexagonal grids. The only difference is that we restrict each ursegment's slope to $|m| \leq \frac{\sqrt{3}}{2}$.

With hexagonal grids, each sweep strip intersects only pixels from a single column. This is in contrast to the sweep strips of the first pass, in which a single strip could intersect two pixel columns. This means the left or right boundary of the sweep strip may be articulated, to conform to the boundaries of the pixels. Each pixel column is comprised of four strips: a strip of left-facing triangles, two strips of square pixels, and a strip of right-facing triangles.

The third sweep is similar to the second. The sweep line is again parallel to a column of pixels, after the axes have been rotated. For square grids, the axes are rotated by an angle of $\pi/4$, so it is equivalent to a vertical sweep; the pixel columns are actually pixel rows with respect to the original axes. This sweep includes exactly those ursegments omitted from the previous sweep, i.e., those with slopes $m < -1$ or $m \geq 1$. For hexagonal grids, the axes are rotated by an angle of $\pi/3$, and the sweep includes ursegments with slopes $m < -\frac{1}{2}$ or $m \geq \frac{\sqrt{3}}{2}$. Note that (i) every ursegment is included in exactly one of the second or third sweeps, and (ii) an ursegment intersects a pixel column boundary in a single point or line segment. This second property avoids any problems with sweeping that might otherwise occur with hexagonal grids, whose pixel column boundaries do not lie on a line.

The hot pixels must be sorted lexicographically by column and row (with respect to the rotated axes) prior to this third sweep.

5.1 The split process

We start this process by reading the next column of hot pixels, and advancing the sweep strip to the start of this pixel column.

We define the following ordering over the ursegments entering the left side of a pixel column with respect to a hot pixel h within the column. If ursegment s enters the pixel column at a pixel below h, and either does not intersect h or first intersects some other hot pixel h^- below h, then $s \prec h$. If s enters the column above h, and either does not intersect h or first intersects some other hot pixel h^+ above h, then $s \succ h$. Otherwise, we say that $s \approx h$. This ordering can be easily adapted for ursegments exiting the right side of a pixel column[4].

We say that bundle b intersects h if $b.lower \approx h$, $b.upper \approx h$, or ($b.lower \prec h$ and $b.upper \succ h$).

For each hot pixel h within the pixel column, we find any bundles that intersect h. We remove each such bundle b found from the tree, and generate three additional bundles: b^-, consisting of those ursegments s within b where $s \prec h$; b', consisting of those s where $s \approx h$, and b^+, consisting of those s where $s \succ h$. We set $b^-.source$, $b'.source$, and $b^+.source$ to $b.source$. If b^- or b^+ are nonempty, we add them to the bundle tree.

If b' is nonempty, we generate an edge between $b'.source$ and h in the rounded arrangement. To test for type (ii) edges, we 'mark' each boundary ursegment s of b' with hot pixel h, which involves the following steps. We set $s.hotPixel$ to

[4]i.e., by reflecting each ursegment and hot pixel involved through the y-axis

h, and test if s intersects hot pixels h^- or h^+ immediately above or below h. If so, we generate the appropriate type (ii) edge.

Observe that when all hot pixels h have been processed, none of the ursegments within the remaining bundles will intersect any of the hot pixels within the column.

The final step in this process is to query the endpoint priority queue and remove any ursegments that are stopping in the current pixel column.

5.2 The sweep process

The sweep process for this algorithm is a simplified version of that of the previous algorithm (section 3.2). In particular, since we have already generated the hot pixels, we do not manipulate any HeatLists, and instead just detect and process ursegment intersections to update the order of the active list. We advance the sweep strip when no intersections remain. We repeat the sweep process for each strip within the current pixel column, to advance the sweep strip to the start of the next pixel column.

To detect type (ii) edges, we take the following steps when an intersection between ursegments s_i and s_j is processed. Let $h_i = s_i.hotPixel$, and $h_j = s_j.hotPixel$. If h_i is defined, and within the current pixel column, we mark s_j with h_i, as described in the previous section. We repeat this procedure for s_i and h_j.

5.3 The create process

The first step in this process is to query the endpoint priority queue and add any ursegments that started in the previous pixel column.

For each hot pixel h in the previous column, we generate a bundle b containing those ursegments s such that $s \approx h$ (according to the 'exit' ordering of Section 5.1). If b is nonempty, we set $b.source$ to h, and add it to the bundle tree.

If more columns of hot pixels remain, the algorithm continues with the split process.

5.4 Performance

Each edge within the rounded arrangement is generated as a type (i) or type (ii) edge within either the second or third sweeps.

To process type (ii) edges efficiently, we store the hot pixels within a sorted, linked list. The test for adjacent hot pixels above or below a hot pixel can then be performed in constant time. For added efficiency, we maintain a flag with each hot pixel to indicate whether a type (ii) edge to its adjacent hot pixel above has already been generated.

During the sweep processes of the three sweeps, we cannot charge the seed events to the snap process, since we are not snapping individual ursegments. Note that an ursegment can occur in at most two seed events within a strip, so the cost of extracting all such events for a pixel row or 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 these terms. As with the first algorithm, once the seed events have been generated, each ursegment intersection can be detected and processed in constant time, for a total cost of $O(|I|)$.

The fact that ursegments will only intersect within hot pixels ensures that the ordering of Section 5.1 is monotonic with respect to a particular pixel h. It follows that when hot pixels are processed during the split processes, each bundle b intersecting a hot pixel can be found and split into b^- and b^+ in $O(\log n)$ time. The time to remove b and insert b^- and b^+ into the bundle tree can be included in this term. The time required to create new bundles during the create processes is similarly $O(\log n)$ per bundle. Since bundles correspond to edges in the rounded arrangement, the time spent in all split and create processes is thus $O(|I^*| \log n)$.

Sorting the hot pixels H at the start of the third sweep requires $O(|H| \log n)$ time. Clearly, $|H| = O(|I^*|)$, so this cost can be charged to the split and create processes.

The total running time of the second algorithm is thus $O(|I| + (|I^*| + \Sigma_c \; is(c)) \log n)$.

6 Conclusion and future work

We have presented two algorithms to perform snap rounding. Both use simple integer arithmetic, are robust, and are practical to implement. They improve upon existing algorithms, since existing running times either include an $|I| \log n$ term, or depend upon the number of segments interacting within a particular hot pixel, whereas ours depend on $|I|$ without the $\log n$ factor and are either independent of the number of segments intersecting a hot pixel (algorithm 1) or depend upon the number of segments interacting in an entire hot row or column, a much coarser partition of the plane (algorithm 2). Ours are the first algorithms to extend snap rounding to hexagonal grids. We have also shown how standard snap rounding cannot be performed on triangular grids (finding a variant of snap rounding that works with such grids is a possible avenue of future research).

Each ursegment processed by our algorithms must have integral endpoint coordinates, so that its endpoints will only occur on strip boundaries. Relaxing this restriction (e.g., by modifying the algorithm to accept endpoints with rational coordinates) is a possible future enhancement.

An applet demonstrating both algorithms is available on the web.[5]

[5]`http://www.cs.ubc.ca/~jpsember`

7 Acknowledgement

We would like to thank William Evans for many helpful comments and suggestions during the preparation of this paper.

References

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

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

[3] M. de Berg, D. Halperin, and M. Overmars. An intersection-sensitive algorithm for snap rounding. *Comp. Geom.: Theory and Appl.*, 36:159–165, 2007.

[4] 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.*, pages 284–293, 1997.

[5] D. H. Greene and F. Yao. Finite-resolution computational geometry. In *27th IEEE Symp. on Found. of Comput. Science*, pages 143–152, 1986.

[6] B. Grünbaum and G. C. Shephard. *Tilings and patterns.* W. H. Freeman & Co., New York, NY, USA, 1986.

[7] L. J. Guibas and D. H. Marimont. Rounding arrangements dynamically. In *COMPGEOM: Annual ACM Symposium on Computational Geometry*, 1995.

[8] D. Halperin and E. Packer. Iterated snap rounding. *Computational Geometry Theory and Applications*, 23(2):209–225, 2002.

[9] J. Hershberger. Improved output-sensitive snap rounding. *Discrete Comput. Geom.*, 39(1):298–318, 2008.

[10] J. D. Hobby. Practical segment intersection with finite precision output. *Comp. Geom.: Theory and Appl.*, 13(4):199–214, Oct. 1999.

[11] L. Middleton and J. Sivaswamy. *Hexagonal Image Processing : A Practical Approach.* Advances in Pattern Recognition. Springer-Verlag UK, August 2005.

[12] V. Milenkovic. Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. In *In Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci*, pages 500–505, 1989.

[13] E. Packer. Iterated snap rounding with bounded drift. In *SCG '06: Proc. 22nd Annu. Sympos. Comp. Geom.*, pages 367–376, New York, NY, USA, 2006. ACM.

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

[15] K. Sugihara. An intersection algorithm based on delaunay triangulation. *IEEE Comput. Graph. Appl.*, 12(2):59–67, 1992.

[16] K. Sugihara and M. Iri. Two design principles of geometric algorithms in finite precision arithmetic. *Applied Mathematical Letters*, 2(2):203–206, 1989.