International Journal of Computational Geometry & Applications

ROUNDING ARRANGEMENTS DYNAMICALLY

LEONIDAS J. GUIBAS
Computer Science Department, Stanford University
Stanford, California 94305, USA
guibas@cs.stanford.edu

and

DAVID H. MARIMONT
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, California 94304, USA
marimont@parc.xerox.com

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

We describe a robust, dynamic algorithm to compute the arrangement of a set of line segments in the plane, and its implementation. The algorithm is robust because, following Greene[1] and Hobby,[2] it rounds the endpoints and intersections of all line segments to representable points, but in a way that is globally topologically consistent. The algorithm is dynamic because, following Mulmuley,[3] it uses a randomized hierarchy of vertical cell decompositions to make locating points, and inserting and deleting line segments, efficient. Our algorithm is novel because it marries the robustness of the Greene and Hobby algorithms with Mulmuley's dynamic algorithm in a way that preserves the desirable properties of each.

Keywords: arrangement, vertical trapezoidal decomposition, dynamic data structure, randomized algorithm, robustness, rounding

1. Introduction

The goal of this paper is to describe a new, robust, and dynamic algorithm for constructing arrangements of line segments in the plane, and its implementation. The problem of constructing line-segment arrangements has been well studied in Computational Geometry and is the focus of some of the most famous algorithms in the field. Let n denote the number of segments we are given and A the complexity of their arrangement (say the number of its vertices). An early solution for this problem was provided by the Bentley-Ottmann sweep,[4] which ran in time $O((n+A)\log n)$; a long sequence of improvements followed, culminating in the optimal but complex $O(n \log n + A)$ algorithm of Chazelle and Edelsbrunner.[5]

The introduction of randomization into Computational Geometry revitalized the problem, and several new randomized incremental algorithms for the problem were invented, all with optimal randomized complexity $O(n \log n + A)$ and notable for their simplicity.[6,7,3] Among these authors, Mulmuley[3] was especially successful at providing an algorithm that was dynamic, allowing efficient segment insertion and deletion.

The key data structure in Mulmuley's algorithm is the vertical cell decomposition (VCD) of a set of line segments. The VCD of a set of line segments is a refinement of their arrangement into cells that are all trapezoids (possibly degenerate) with two vertical sides. As is well established by now, there is a large class of applications in Computational Geometry for which this further refinement of the arrangement into cells each with only a small number of sides (four in our case, assuming non-degeneracy) is very useful. The Mulmuley algorithm is randomized and dynamic: that is, it is possible to insert and delete line segments from the VCD. The output of the algorithm is a hierarchy of (VCDs) vertical cell decompositions of subsets of the line segments. The lowest, most detailed level contains all the line segments. Each higher level contains a randomly chosen subset of the line segments present one level down. The hierarchy makes it possible to locate points, insert and delete line segments, and otherwise navigate efficiently around the arrangement.

This, as well as all previously mentioned algorithms, were developed assuming an infinite-precision model of computer arithmetic. In any practical implementation of a line segment arrangement computation, however, the implementors have to consider the effects of finite precision arithmetic on the above techniques. This issue of robust implementation of geometric algorithms has been addressed in several papers,[8,9,10,11,12,13] but with mixed success. We do not attempt to survey this extensive literature here. In our work, we will address the robustness problem by perturbing all vertices of our arrangement to lie on a fixed grid, assumed for convenience to be that of the points with integral coordinates. However, in order to ensure that the perturbed arrangement has a topology consistent with the original, we will need to perturb to the grid additional features of our arrangement as well. In the end, all vertices, edges, and faces of our perturbed arrangement will have exact representations with finite arithmetic. We call this operation *rounding* the arrangement. Exactly how to accomplish such a perturbation of the arrangement to the grid was first studied by Greene and Yao.[14] As we explain below, the Greene-Yao rounding has a number of undesirable properties, which were overcome in another rounding scheme proposed by Greene, and independently by Hobby, in as yet unpublished manuscripts.[1,2] A still third way to round, called *shortest path rounding* was proposed by Milenkovic;[15] see his paper and the references cited therein.

The key contribution of our paper is to show how to combine the ideas of Mulmuley's dynamic segment arrangement algorithm, while maintaining (and producing) only the rounding of Greene and Hobby of the arrangement of the current segments. The algorithms proposed by Greene and Hobby are robust but not dynamic; they are based on finite-precision arithmetic but operate in a batch mode that assumes all the segments are given at once. The Mulmuley algorithm is dynamic but not robust; it provides for inserting and deleting line segments but is based on infinite-precision arithmetic. The technical challenge we have to overcome is how, using only the VCD of the rounded arrangement of the present segments (and its hierarchy of sampled counterparts), to simulate the effect of doing an insertion or deletion in the ideal Mulmuley structure and then rounding the result. We guarantee that the rounded arrangement we compute dynamically is exactly the same as what would have been produced by the batch algorithms of Greene and Hobby. We have implemented and extensively tested this new algorithm.

In the paper we begin by describing the rounding of Greene and Hobby and a new and elementary way to derive its desirable topological properties (Sections 2 and 3). We then provide a succinct summary of our data structures and discuss the key algorithmic issues in adding a new segment to the arrangement (Sections 4 and 5). Section 6 discusses deletions and the effect of the hierarchy on the algorithm. In Section 7 we provide a brief analysis of our method. In Section 8 we talk about some of the experiences from our implementation. We end by presenting some conclusions in Section 10.

2. Snap Rounding

In this section we briefly discuss a way to round an arrangement of line segments that is especially economical in terms of the number of "kinks" introduced in the segments. This method, as already mentioned, was introduced by Greene[1] and Hobby[2] — we shall refer to it from here on as *snap rounding* for reasons that become apparent below.

The setting is as follows: the Euclidean plane is tiled into unit squares each centered on a point of the integer grid, in the obvious way; we refer to these tiles as *pixels*.[a] As mentioned, we coordinatize the plane so that pixel centers have integral coordinates and refer to these pixel centers as *integral* points. When we round an arrangement of line segments, we require that all its vertices (endpoints of segments, as well as intersections of pairs of segments) be perturbed to integral points — in other words, the only points we allow as vertices in our rounded representation are the integral points.

Any rounding scheme must have at least two goals: (1) to keep the perturbed segments near the originals, and (2) to preserve as much as possible the topology of the original arrangement. Requirement (1) suggests that each vertex of the ideal arrangement be perturbed to its nearest integral point. However, it is well known that just doing this can cause topological inconsistencies between the ideal and the rounded arrangements. In order to avoid this problem, about eight years ago, Greene and Yao[14] suggested that we treat each representable point as an "obstacle" and do not allow our segments to go over these obstacles while vertices move to their nearest integral point. These obstacles can create additional "kinks" in the perturbed segments. Greene and Yao showed that, with this additional fragmentation, no topological inconsistencies arise. They also gave an algorithm for

[a]These pixels correspond to a tiling of the plane at any desirable resolution and need not correspond to the size of the display pixels of an output device.

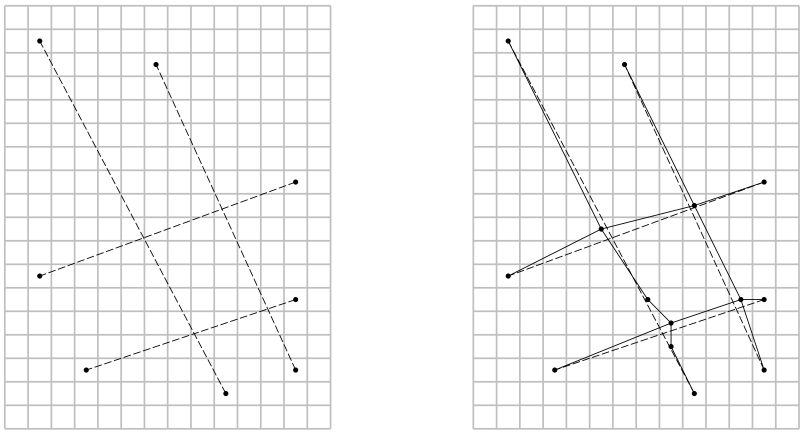

Figure 1: Left, a small line segment arrangement; right, its Greene-Yao perturbation.

computing their perturbation efficiently.

The difficulty with the Greene-Yao method is that the requirement of not going over the obstacle grid adds a large number of additional breaks to each rounded segment. See, for example, Figure 1(left) showing an ideal line arrangement (dashed lines), and Figure 1(right) its Greene-Yao perturbation (solid lines). In that figure the grid lines (gray lines) correspond to pixel boundaries.

To get around the excess fragmentation, snap rounding proceeds as described below. To fix the terminology, we call the original unrounded segments *ursegments*. After the perturbation, each ursegment becomes a polygonal line that we call a *polysegment*. A polysegment consists of smaller line segments which themselves are called *fragments*. If s denotes an ursegment, we denote the corresponding polysegment by the corresponding Greek letter σ. For brevity of exposition, we assume that every vertex of our arrangement has a unique nearest integral point — fortunately, such degeneracies do not introduce any substantial difficulties.

We declare all pixels containing an ursegment endpoint, or the intersection point of two ursegments, to be *hot*. In other words, any pixel containing a vertex of the ideal arrangement becomes hot — note that a pixel may become hot for multiple reasons. Snap rounding is then this: if an ursegment terminates in a hot pixel, it is perturbed to terminate at the that pixel's center; and if an ursegment passes through a hot pixel, then it is perturbed to pass through that pixel's center. See Figure 2 for an illustration. Notice this key aspect of snap rounding: a kink is added to an ursegment only where a vertex of the arrangement lies on the segment, or where the ursegment passes "very near" an integral point which will become a vertex of the rounded arrangement. Figure 2(right) shows snap rounding for the arrangement of Figure 1(left) — it is evident that fewer kinks have been added. We call this snap rounding because all ursegments passing through a hot pixel are snapped to pass through that pixel's center.

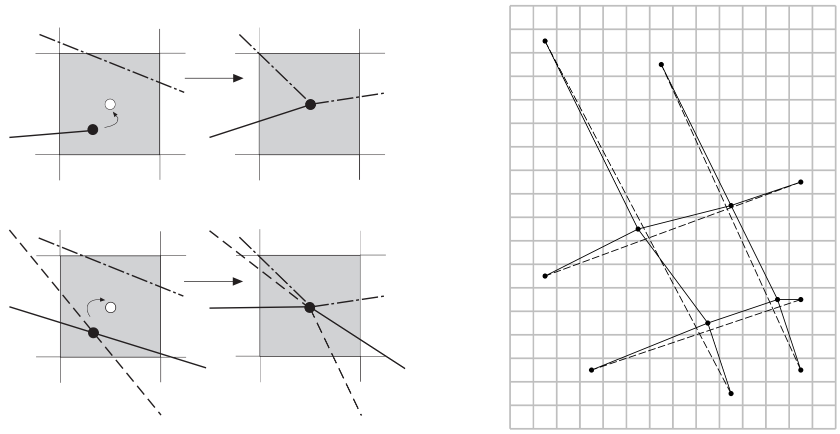

Figure 2: Left; the rules of snap rounding; right, the snap-rounded form of the arrangement in Figure 1(left).

The shortest path routing of Milenkovic[15] was introduced for performing robust boolean operations on polygons. It is based on the idea of continuously perturbing all arrangement vertices to their nearest integral point while at the same time detecting when we encounter other fragments and then dragging them along (to prevent topological violations). Shortest path rounding adds even fewer kinks than snap rounding but it seems more complex to compute and maintain, as it is defined by a continuous process and therefore lacks a simple input-output specification.

Though all three rounding approaches presented above are defined in terms of modifications to the ideal arrangement, it is conceivable that they can be computed and maintained directly from the input set of segments. Since in practice a rounded arrangement can have far lower complexity that its ideal counterpart, algorithms that compute the rounded structure directly, bypassing the ideal arrangement, can be more efficient. We have selected to work with snap rounding because of its clean mathematical properties; these have recently led to a direct output-sensitive algorithm for the rounded arrangement.[16]

3. A Topological Analysis of Snap Rounding

It is first of all trivial to prove that after snap rounding, an ursegment and the corresponding polysegment are quite near each other.

Lemma 1 *After snap rounding, the polysegment σ corresponding to ursegment s is contained within the Minkowski sum of s with a pixel (unit square) centered at the origin.*

Proof. The hot pixels crossed by ursegment s can be linearly ordered in the sequence in which they are crossed by s. Consider now the part of s in and between two successive hot pixels p_1 and p_2 in this sequence. From the central symmetry of the pixel shape, it follows that the Minkowski sum of s with a pixel contains the pixel centers of p_1 and p_2. But this Minkowski sum is obviously convex, so it

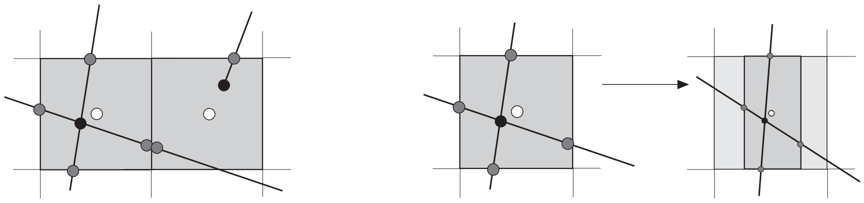

Figure 3: Left, segment fragmentation by nodes at hot pixel boundaries; right, halfway through the first stage of the explicit deformation.

also contains the fragment of the corresponding polysegment σ joining those two centers. □

Understanding the relation between the topology of the original and rounded arrangements is much more interesting. In his original manuscript, Greene defined and proved a number of "topological consistency" properties of his perturbation. His argument was phrased in a context allowing more general pixel shapes than we have here, but it was involved and required a global analysis using graph-theoretic arguments. In this paper we give a very different topological analysis of snap rounding. We show topological consistency between the original and rounded arrangements by giving an *explicit continuous deformation* that takes an arrangement of segments into its snap-rounded form. During that deformation features of the arrangement may collapse (as they must in any rounding scheme), but they never invert — in the sense that no vertex of the arrangement ever crosses through one of the segments. We call the latter property the *non-penetration condition*. Our deformation is easy to visualize and the proof of non-penetration is entirely local and straightforward. In addition, because of the clear understanding of this topological transformation that our deformation provides, we are able to easily prove certain additional lemmata that are useful in the implementation of our incremental variant of snap rounding.

We now proceed to define a deformation $\mathcal{D}$ that starts from the original arrangement and continuously transforms it to its snap-rounded form. We initially break up each ursegment into a number of subsegments, by introducing a breakpoint, or *node*, whenever the ursegment crosses the boundary of a hot pixel. Note that if an ursegment crosses a pixel boundary separating two hot pixels, then two nodes will be placed there, with a zero-length subsegment between them. See Figure 3(left) for an illustration.

Our deformation $\mathcal{D}$ proceeds in two stages. In the first stage every hot pixel contracts simultaneously and at the same rate in the x direction, towards the vertical axis through its center. If at time $t = 0$ we have the original arrangement, then each hot pixel is linearly scaled down in the x-direction until, at time say $t = 1$, all hot pixels have collapsed to their vertical axis. All nodes on the hot pixel boundaries follow the pixel motion. Each ursegment thus is continuously deformed to a polysegment, the polysegment joining the current locations of its nodes for each time t. At time $t = 1$ each hot pixel has become a hot "stick," with nodes marked on it. In the second stage of the deformation, say from $t = 1$ to $t = 2$, the hot sticks vertically contract simultaneously and at the same rate towards their center. Again, the polysegments are defined by just tracking the corresponding nodes. See Figure 3(right) for an illustration of this process.

It is not hard to show that at time $t = 2$ our polysegments are exactly those defined by snap rounding. Note that, during the deformation $\mathcal{D}$, our nodes partition each polysegment into fragments of two kinds: *internal* — those inside a hot pixel, and *external* — those connecting nodes on the boundary of two different hot pixels. At the end of the deformation $\mathcal{D}$, internal fragments collapse to pixel centers and only external fragments remain. No node ever crosses over a fragment during this deformation. In particular, an endpoint of a polysegment can never move over another polysegment, nor does the orientation of an elementary triangle formed by three polysegments ever invert — see Figure 4.

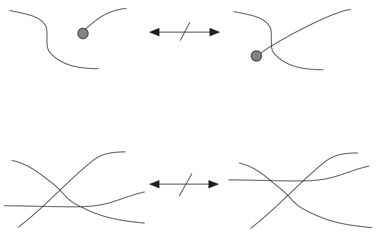

Figure 4: Some of the topological consistency conditions.

Theorem 1 *During the deformation $\mathcal{D}$, no node ever crosses over a fragment.*

Proof. It is first of all clear that there can be no trouble among all the internal fragments of a particular hot pixel: they all occur in a region of the plane that undergoes a uniform x- or y-scaling transformation. The crucial part of the proof is the invariant that no external fragment ever touches or enters a contracting hot pixel (other than those at its endpoints). To see this, we can argue as follows. Consider, for example, the x-part of the deformation (first stage). If an external fragment is to enter a hot pixel, then it must overtake during the deformation one of the corners of that pixel's boundary. Furthermore, at the point of overtaking, the external fragment and the corner must be moving in the same direction in x (i.e., both left or both right), as the hot pixel is contracting. But by the way we have defined the x-deformation, the x-velocity of the pixel corners is always maximal in magnitude among all points on a pixel boundary. And in addition, a point along a fragment is moving in x by a velocity which is a convex combination of the velocities of its endpoints, which are nodes following other hot pixel boundaries. An entirely analogous argument holds for the y-part of the deformation. Thus a fragment can never overtake a deforming hot pixel. □

As a consequence of the above, we can deduce many topological consistency properties between the original line segment arrangement and its snap rounding.

Corollary 1 *After the snap rounding deformation $\mathcal{D}$:*

a. No fragments intersect except at their endpoints.

b. The circular ordering of the fragments ending at the same pixel is consistent with the circular ordering at the pixel's boundary of the corresponding ursegments crossing this pixel.

c. If ursegment r intersects ursegment s and then ursegment t, then polysegment ρ cannot intersect polysegment τ before polysegment σ.

d. If a vertical line ℓ through pixel centers intersects ursegment s and then ursegment t, then ℓ cannot intersect polysegment τ before polysegment σ.

Proof. These claims are immediate.

a. At the beginning of the deformation no external fragments properly cross, and this property is maintained throughout $\mathcal{D}$ because of the above theorem. Furthermore, all the internal fragments collapse to points at the end of $\mathcal{D}$.

b. The hot pixel boundaries deform continuously, and so the ordering of the nodes around them is preserved during $\mathcal{D}$.

c. An ursegment r and its corresponding polysegment ρ go through hot pixels in the same sequence; in particular this applies to the hot pixels created by the intersections of r with s and t. When these intersections lie in the same hot pixel, snap-rounding forces them to coincide.

d. For such an inversion to happen, ursegments s and t would have to intersect, and snap rounding would have to move the intersection across a vertical line though pixel centers — which obviously cannot happen.

□

4. Data Structures

Here we briefly summarize the data structures we use. The vertical cell decomposition's top-level data structures are vertices and two types of line segments, fragments and vertical attachments. Figure 5(left) illustrates these structures; a vertex is represented by a filled circle, a fragment with a solid line, and a vertical attachment by a dotted line with an arrow that points to the fragment at one end of the attachment. Note that the 'x'-shaped configuration of fragments at the top of the figure is not two fragments intersecting at a vertex, which is impossible by corollary 1, but four fragments that share an endpoint. For convenience, we bound the VCD with a rectangle of four ursegments with integral endpoints, as is standard.

An ursegment is stored with the fragments that make up its polysegment. Snap rounding can cause two or more ursegments to share the same fragment, so each fragment maintains a list of ursegments that it represents, sorted in their y-order (this is well defined because two ursegments cannot cross between the two hot pixels delimiting the fragment). Although we can concoct pathological situations where all or most segments share some fragments, experiments with both randomly generated line segments and edge data from natural images suggest that these lists rarely have

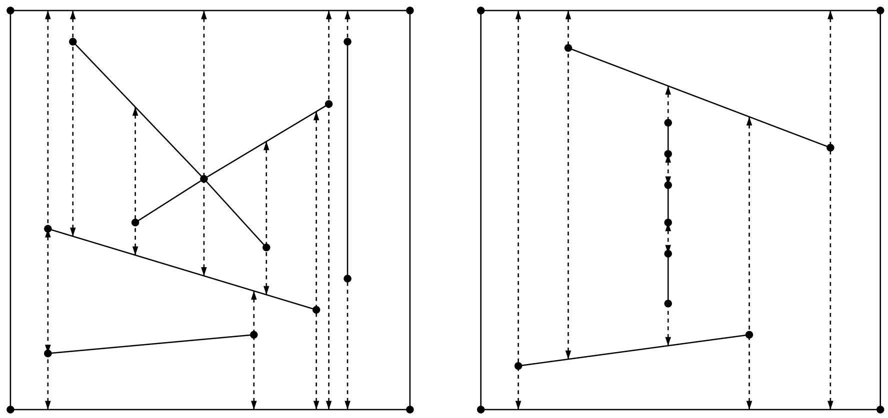

Figure 5: Left, top-level structures in the VCD; right, a degenerate vertical boundary.

more than five ursegments. An ursegment also has pointers to the vertices at its endpoints.

Snap rounding can produce "degenerate" configurations that the VCD must be able to represent, such as the vertical fragment in the upper right of Figure 5(left). Such degenerate configurations can make the vertical boundaries of a trapezoid arbitrarily complicated, as shown in Figure 5(right).

The VCD's data structures make it easy to move vertically. Moving vertically from a vertex, vertical fragment, or vertical attachment is straightforward because each can have only one structure above and one below. Nonvertical fragments, however, can have a sequence of structures above and below them, which we represent with two doubly linked lists called the *ceiling* (for the structures above) and *floor* (for those below) lists of the fragment. In addition to facilitating vertical motion from a nonvertical fragment, they make it easy to move along the fragment.

Figure 6 illustrates the ceiling and floor lists of some nonvertical fragments in a simple VCD. On the left is a VCD with four nonvertical fragments. We shall refer to a fragment by the number that appears to the left of its left endpoint. Figure 6(right) shows the ceiling and floor lists of the fragments schematically. Each fragment f is represented by a horizontal line segment with filled circles at its endpoints; the number of fragment f appears to the left of the line. The numbers and arrowheads above f's line correspond to the fragments and vertical attachments above f as we move from its left endpoint to its right. Pointers to these fragments and vertical attachments are stored on the ceiling list of f. The numbers and arrowheads below f's line correspond to the fragments and vertical attachments below f, pointers to which are stored on the floor list of f.

The vertical attachments on a floor or ceiling list define a partition of the interval in x occupied by the fragment. The open intervals in each list, where another fragment is above or below the fragment, we represent with structures called *xspans*.

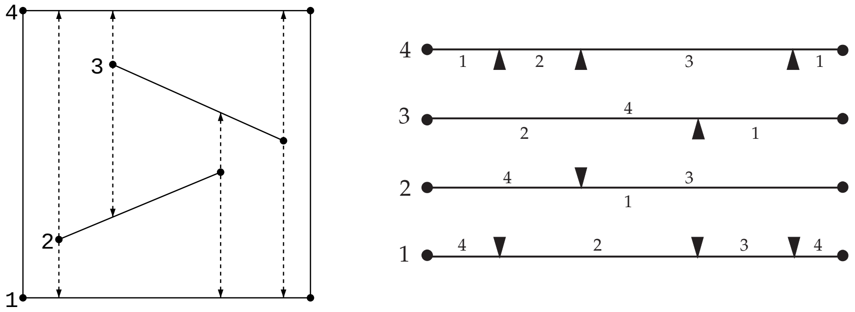

Figure 6: Left, a simple VCD; right, the fragments' ceilings and floors.

Each trapezoid in the VCD is bounded below by a ceiling xspan (belonging to the fragment at the bottom of the trapezoid) and above by a floor xspan (belonging to the fragment at the top). For example, the trapezoid at the center of Figure 6(left) is bounded below by the second xspan on 2's ceiling list, and above by the first xspan on 3's floor list. The floor and ceiling xspans that bound a trapezoid point to each other, which makes it possible to move vertically across a trapezoid. To cross from the floor of a nonvertical fragment to its ceiling (or vice versa), we maintain with each fragment a third doubly linked list consisting of all vertical attachments on the floor and ceiling lists.

A simple traversal algorithm for locating a point illustrates how these structures are used. We first find the xspan containing the x coordinate of the point by linearly searching the ceiling list of the horizontal fragment that defines the bottom of the VCD's bounding rectangle. We search upwards in y, using xspans to cross trapezoids, for example, until we find the trapezoid that contains the point. We discuss an algorithm for locating points that exploits the hierarchy of VCDs in Section 6.

5. Inserting a New Ursegment

Inserting a new ursegment s requires three different searches. First, we must determine the new hot pixels created by s by locating its endpoints and detecting its intersections with other ursegments. Second, we must detect the existing hot pixels through which s passes. Third, we must detect the existing ursegments that pass through the new hot pixels (and perturb the ursegments accordingly).

The polysegment σ of s is defined by the vertices of the new and existing hot pixels found in the first two of these searches. Once the vertices of σ are known, we insert those of its fragments not already in the VCD (with the new ursegment the only member of its list of ursegments); by corollary 1 these fragments intersect no others, which makes the insertion simple. For a fragment of σ already in the VCD, we need only add s to its list of ursegments. Note that the order in which σ passes through these pixel centers can be immediately determined from the slope of s.

Figure 7 illustrates the problem of detecting s's intersections with existing ursegments. The ursegments are the dashed lines, the fragments are the solid ones, and the vertices are filled circles. The grid of pixel boundaries are the gray lines. The new ursegment s is the near-horizontal dashed line that begins at the left of the figure. Three existing ursegments and their polysegments are shown.

Each ursegment and its polysegment define a closed but not necessarily simple polygon. We call these polygons *slivers*. The existing ursegments at the left and the center of the figure depict the most common situation. In each case, s passes entirely through a sliver, so that s intersects both the existing ursegment and one of its fragments. (In general, s may intersect several of the fragments.) Detecting intersections with such ursegments is easy, since we need only find intersections with fragments, and test the ursegments to which the fragments belong.

The rightmost existing ursegment in the figure depicts a more challenging situation that can only arise at an endpoint of s. Here s intersects the existing ursegment r, but because an endpoint Q of s lies inside the sliver, s does not intersect any of the fragments of ρ, the polysegment of r. In this case the endpoint Q will define a new hot pixel h and either r or ρ (or both) must intersect h. Note that if r does not intersect h, then the center of the hot pixel h must lie outside the sliver and thus ρ must separate it from Q. So in all cases either r intersects h, or ρ intersects the segment from Q to h's center.

To find the ursegments that intersect s, we use the following algorithm, which we call *VCD traversal*. First, we locate the left endpoint P of s. We then walk, using the VCD, in a straight line from P to the center of the pixel h in which it lies and collect all fragments thus encountered. For each of them we test the associated ursegments to see if they intersect s. We also use the vertical range search outlined below to collect all ursegments intersecting h and test each of them for intersection with s. We then follow s itself through the VCD. Whenever s intersects a fragment, we test the fragment's ursegments to see whether they intersect with s. Finally, when the right endpoint Q of the new ursegment s is reached, we perform tests analogous to those at the left end. The new hot pixels are those containing the endpoints of s and its intersections with existing ursegments.

The second of the three searches is for existing hot pixels through which s passes. Let the *sausage region* of s be the Minkowski sum of s with a pixel centered at the origin. If s passes through a hot pixel, the vertex at the pixel's center must lie in the sausage region of s. To find the hot pixels through which s passes, we visit each cell that intersects the sausage region and test whether the vertices that lie on the cell boundary are inside the sausage region. We call this search *cell sweep*.

The algorithm for cell sweep is as follows. We find the list of cells that intersect s; call this the sweep list. The idea is to sweep upwards from these cells to the top of the sausage region, then downwards to the bottom. To sweep upwards, we mark each cell on the sweep list and test whether each hot pixel center on the left boundary of the cell is inside the sausage region. If so, we put it on a list of hot pixels. We remove the first cell from the sweep list; call it cell i. We replace it on the sweep list with a list of the cells above cell i that have not yet been marked, and mark these cells. For cell j to be above cell i, part of cell j's bottom boundary

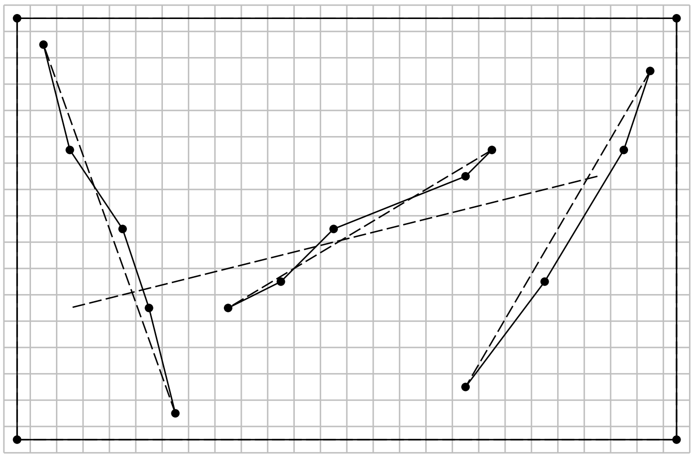

Figure 7: Detecting a new ursegment's intersections with existing ursegments.

must coincide with part of cell i's top boundary. We exclude from the sweep list cells that are completely outside the sausage region. When a cell is placed on the sweep list, we test whether the hot pixel centers that lie on its left boundary are inside in the sausage region and if they are, we add them to the hot-pixels list.

We keep removing the first cell on the sweep list and replacing it with the ones above until the sweep list is empty. We then perform an analogous sweep downwards. We unmark all marked cells and return the list of hot pixels. These are the hot pixels through which s passes. It is clear that the cost of the cell sweep is proportional to the number of cells that intersect the sausage region of s.

The third search is for existing ursegments that pass through a new hot pixel. Figure 8 shows an example of this search at one hot pixel. The top left panel shows the VCD at the beginning of the search; the hot pixel is the shaded square near the center. This VCD consists of two ursegments (aside from those making up the bounding rectangle), each with a single-fragment polysegment. Each ursegment coincides with its fragment because it has integral endpoints. Both ursegments pass through the new hot pixel. The first step is to insert a vertex at the hot pixel. This is shown in the top right panel. The new vertex does not lie on a fragment, and it has vertical attachments extending up and down to the vertically adjacent fragments. (Such isolated vertices are another degenerate configuration supported by our data structures.)

To find these ursegments, we perform a *vertical range search.* First, we search up the vertical attachment extending above the new vertex. Whenever a fragment is encountered, its ursegment is tested to see whether it passes through the hot pixel. If it does, the fragment is perturbed to pass through the new vertex. This perturbation is illustrated in the left panel of Figure 8. Here an ursegment s on the ursegments list of the fragment f above the new vertex does pass through the hot pixel; in this case $s = \sigma = f$. In the bottom left we see that the fragment f has

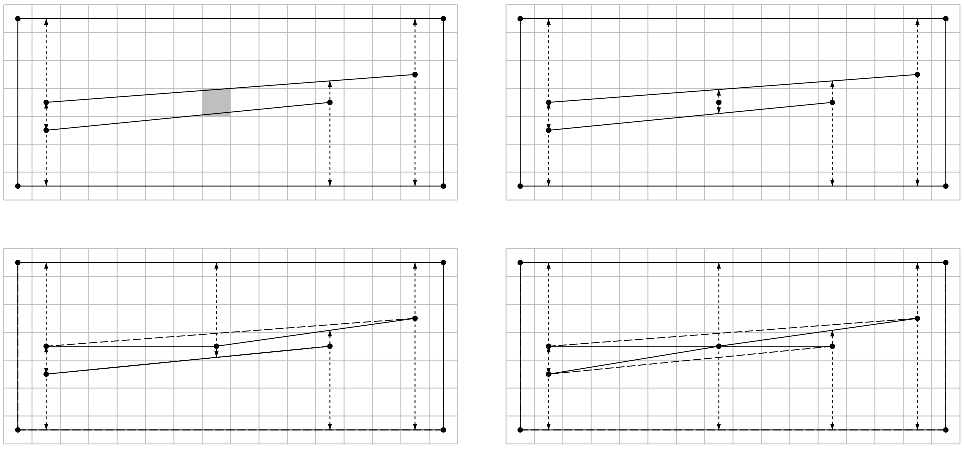

Figure 8: The search for existing ursegments that pass through a new hot pixel.

been split in two, so that σ now passes through the new vertex; we see the original ursegment s as a dashed line. The vertical attachment above the new vertex now points to the fragment above the one that was just perturbed. In this case, this fragment is the upper boundary of the VCD's bounding rectangle. Its ursegment does not pass through the new hot pixel, and therefore the search upwards can be terminated. This algorithm works because, by corollary 1, snap rounding ensures that the order of fragments along any vertical line through pixel centers contains no reversals of the order of the ursegments to which they belong. An analogous search is performed downwards. The bottom right panel of Figure 8 shows the VCD after the search downwards has terminated.

6. Deletions and the Hierarchy

To delete an ursegment, we visit the fragments of s's polysegment σ and remove s from each fragment's list of ursegments. When a list becomes empty, we delete the fragment. Next, we delete each vertex of σ in a hot pixel that became hot only because s ended or intersected another ursegment in the pixel; this can be decided efficiently by using the order in which the ursegments enter or leave that pixel. (This order is available to us by combining the ordering of the fragments around each vertex and the ordering of the ursegments within each fragment.) These operations leave the VCD in the state in which it would have been had s never been inserted.

As mentioned earlier, our algorithm produces a hierarchy of VCDs, where the lowest level (level zero) contains all the ursegments, and each successively higher level contains a subset of the ursegments present one level down. Adjacent levels are linked through their hot pixels and ursegments; each hot pixel or ursegment at level i has a *descent* pointer down to the corresponding hot pixel or ursegment at level $i-1$.

To locate a point at the bottom level of the hierarchy of VCDs, first we locate it in the top (least detailed) VCD in the hierarchy, say at level i, using the point location algorithm described in Section 4. Next, we find a nearby hot pixel and use its descent pointer to locate the corresponding hot pixel in the VCD one level down, at level $i-1$. (This nearby hot pixel is never more than a pointer or two away, because every cell has at least two hot pixels on its boundary.) To locate the point from a vertex at level $i-1$, we trace a straight line through the VCD from the hot pixel to the point. We repeat this process until the bottom level of the hierarchy has been reached.

A new ursegment is inserted into the l bottommost levels of the hierarchy, where l is computed independently for each ursegment by flipping a coin until tails is obtained. To insert the ursegment, we locate an endpoint at all levels of the hierarchy as above, and then insert the ursegment at the bottommost l levels independently using the algorithm described in Section 5. At each level, the descent pointers of the ursegment and of any new vertices created by the insertion are linked to the corresponding ursegments and vertices one level down.

Figure 9 depicts a hierarchy of VCDs associated with an arrangement of 14 ursegments. Each row in the figure shows one level of the hierarchy, with ursegments on the left and fragments on the right. The top row is the top of the hierarchy and contains only one ursegment; the other rows contain two, nine, and fourteen respectively.

An ursegment is deleted from the hierarchy by deleting it independently from each level.

These algorithms are quite similar to those proposed by Mulmuley.[3] In the Mulmuley algorithm, of course, there are no hot pixels, so there is no need to decide whether a vertex is still in a hot pixel after deleting an ursegment, as we must.

7. Analysis of the algorithm

Let n be the number of ursegments we have and let A denote the combinatorial complexity (say, the number of vertices) of their ideal arrangement $\mathcal{A}$. It is well known that Mulmuley's randomized incremental algorithm[3] builds $\mathcal{A}$ in expected time $O(n \log n + A)$. Let $\mathcal{R}$ denote the rounded arrangement of these same segments and let R denote the combinatorial complexity of $\mathcal{R}$; it is clear that $R \leq A$. The reduction in complexity from $\mathcal{A}$ to $\mathcal{R}$ is due to the fact that in the latter several features (e.g., vertices) collapse to the same feature. It is useful to think of R as follows. Let $\mathcal{H}$ denote the set of hot pixels; for a hot pixel h, let $|h|$ denote its combinatorial size, i.e. the number of ursegments passing through it. Then it is clear that

$$R = O(\sum_{h \in \mathcal{H}} |h|). \tag{1}$$

(Note that R can actually be less, as when many tiny segments appear within the same hot pixel). In our representation of the VCD, we store with each fragment a list of all the ursegments that gave rise to it. The additional storage required for these lists is easily bounded by $O(\sum_{h \in \mathcal{H}} |h|)$ as well. We observe that the latter sum can be either larger or smaller than A (the sum of hot pixel sizes can exceed the true arrangement complexity if many ursegments cross through many hot pixels,

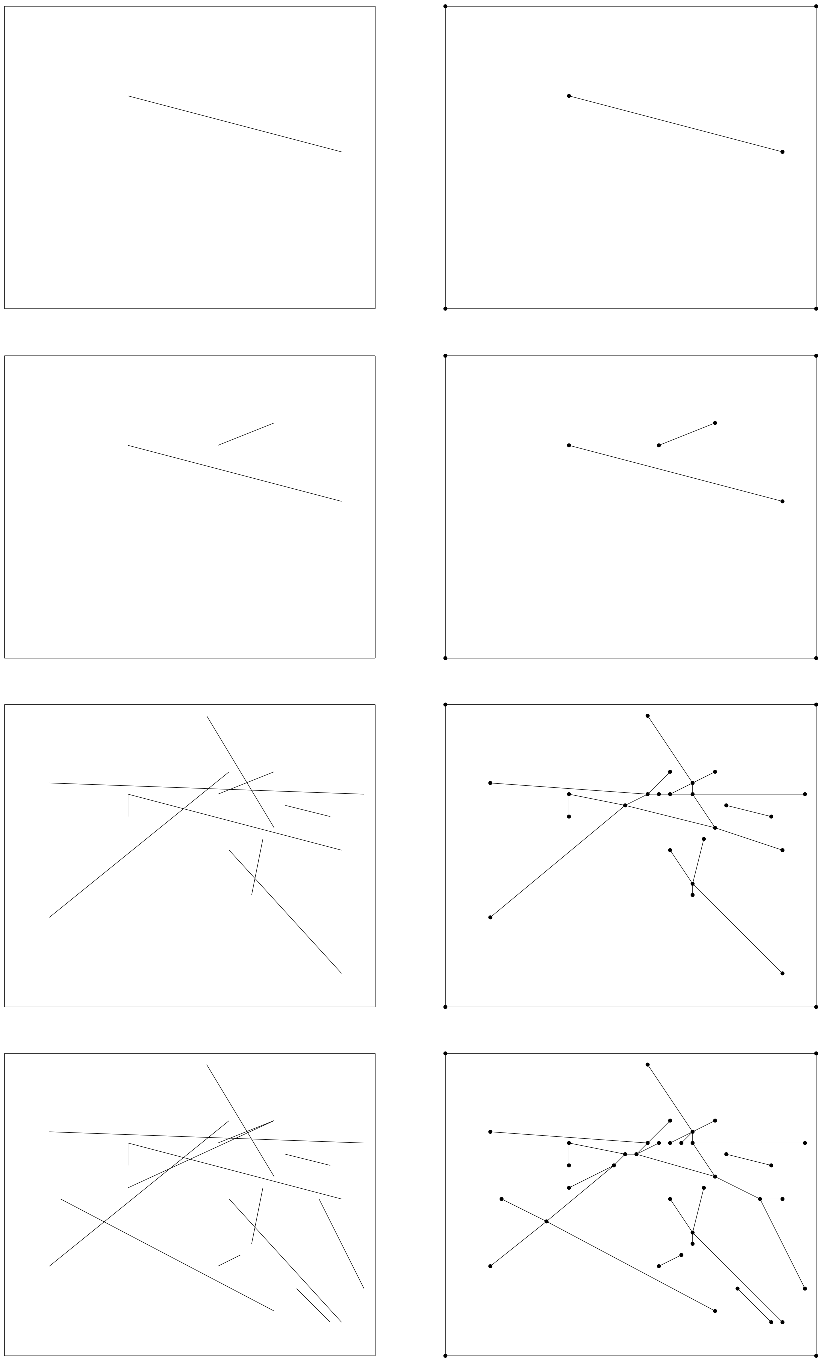

Figure 9: A hierarchy of VCDs: the rows are the levels of the hierarchy from top to bottom. The left column shows the ursegments, and the right column the fragments.

but without giving rise to features of $\mathcal{A}$ there). For our analysis we also need the following quantity: define a pixel as being *warm* if it contains the endpoint of a vertical attachment in the ideal VCD of $\mathcal{A}$. We let $\mathcal{W}$ denote the set of warm pixels and $\mathcal{W}'$ the set of pixels that are either warm or are neighbors (king-wise) of warm pixels. Let $C = \sum_{w \in \mathcal{W}'} |w|$ be the total complexity of these 'near-warm' pixels: this quantity is used to bound the complexity of the cell sweeps.

The most delicate issue in the analysis of our incremental snap rounding algorithm is the effect of pixel size on performance. If the pixel size is extremely small compared to separation of the vertices in $\mathcal{A}$, then we expect our algorithm to behave the same as the ideal randomized algorithm working over the reals. As the pixel size gets larger, two opposite effects come into play. On the one hand the VCD becomes coarser and its size can drop significantly — and so can the cost of traversing it. On the other hand, we lose any knowledge of the structure of $\mathcal{A}$ *within* the now large hot pixels, and as a result discovering intersections between existing ursegments and a newly added one can become expensive. In particular, when a few pixels cover all of $\mathcal{A}$, our algorithm naturally reduces to the naive quadratic algorithm that checks all pairwise ursegment intersections though, as mentioned earlier, better methods have recently been found.[16]

Consider an existing polysegment σ and its corresponding ursegment s. As we already remarked, the vertices of σ define a sequence of hot pixels which both σ and s pass through in the same sequence. The following obvious lemma is critical for our analysis.

Lemma 2 *During the incremental construction, if a new ursegment t intersects the fragment f of an existing polysegment σ between hot pixels h_1 and h_2, then one of the following three situations holds:*

a. t also intersects s, the ursegment corresponding to σ, between (and outside) h_1 and h_2, or

b. t terminates in the sliver between s and σ, or

c. t enters at least one of the hot pixels h_1 and h_2.

Proof. This is because s, together with f and the two pixels h_1 and h_2 define a closed region entered by t. □

This lemma allows us to estimate the cost of the VCD traversal for a new ursegment t. The cost of this search, over and above that of the ideal Mulmuley algorithm, is in checking for the possible but non-existing or already discovered intersection between t and other ursegments associated with fragments crossed by t. It is clear that t may cross several of the fragments defining σ, while it can cross s only once, or not at all. Case (a) of the above lemma is the favorable one — where a fragment crossing has a corresponding ursegment crossing; this is paid for by A. In case (b) the endpoint of t will will require the local search through the fragments around the hot pixel vertices of the its surrounding trapezoid in the VCD, as explained in Section 5. Finally note that each time case (c) holds, ursegments s and t must pass though the same hot pixel (h_1 or h_2).

Another difference between our situation and that of Mulmuley is that we cannot assume non-degeneracy. We may have trapezoids in the VCD whose vertical sides are not just vertical attachments: a particular side can be an arbitrarily long alternation of vertical fragments and vertical attachments. Thus we can no longer assume that traversal through a trapezoid in a constant-time operation. We can deal with this problem by using the Mulmuley hierarchy not only for end-point location but for the VCD traversal itself as well. In other words, when a new ursegment s is given, we 'drop it' through the VCD hierarchy starting from its left endpoint and proceeding to the right, letting the hierarchy navigate us though such complex vertical sides. It is not difficult to estimate the cost of this method and show that it leads to the same asymptotic bounds as the standard Mulmuley analysis[b].

These observations lead to the following analysis for our algorithm; note that $C = \sum_{w \in \mathcal{W}'} |w|$, the quantity introduced earlier, bounds the total number of ursegments (with multiplicity) that pass within half-a-pixel above or below the non-hot-pixel endpoints of all vertical attachments.

Theorem 2 *The complexity of our incremental rounded arrangement construction procedure is*

$$O(n \log n + A + \sum_{h \in \mathcal{H}} |h|^2 + C). \tag{2}$$

Proof. The key is to understand the insertion cost of a new ursegment t. The above paragraph takes care of the VCD search for the intersections between t and other existing ursegments t; these intersections, together with the endpoints of t define the new hot pixels created by t. The total cost of these searches, summed over all insertions, is captured by the bound in the theorem. To see this we argue as follows. Events of type (a) in the above lemma can be charged to vertices of the ideal arrangement, and thus can be paid out of the A term in our bound. An event of type (b) requires us to look at all fragments passing between an endpoint of t and the center of the hot pixel h it lies in, as outlined in in Section 5. Such fragments whose corresponding ursegments intersect t can also be paid out of the A term. Any other such fragments must have an ursegment that intersects h. These fragments, as well as all those examined in case (c) of the above lemma, can be paid out of the term $\sum_{h \in \mathcal{H}} |h|^2$ in the theorem. This is so because they all correspond to a unique pair of ursegments passing through the same hot pixel (h, h_1, or h_2 as the case may be).

Any existing ursegments crossing the new hot pixels need also to be snap-rounded to these new pixel centers. The vertical range search method for this given in Section 5 takes time proportional to the number of such ursegments (plus one, to be exact), as the search (down or up) stops as soon as we encounter a non-crossing ursegment. The total cost for these searches is then $O(\sum_{h \in \mathcal{H}} |h|)$.

It remains to deal with the cell sweeps. During the insertion of t, the cell sweep finds any existing hot pixels crossed by the new ursegment t. Without loss of generality, we confine our attention to the upwards sweep; the analysis for the

[b] In our current implementation we do not do this; we simply walk along a vertical side to find where the segment currently being added crosses it.

downwards one will be symmetric. We remarked in Section 5 that the cost of this cell sweep is proportional to the number of trapezoids contained in the sausage of t. Note that each trapezoid of the VCD has at least two vertices which are hot pixels. If a trapezoid is fully contained in the sausage of t, then at least one of its vertices is a hot pixel center, and t crosses that hot pixel. This leaves unaccounted for trapezoids which partially intersect the sausage and all of whose vertices in the sausage are endpoints of vertical attachments. It is easy to check that in that case each such vertex must be within a 1-neighborhood of some warm pixel, and that the ursegment t must intersect this neighborhood. Thus such vertices and their trapezoids are paid for by the term $C = \sum_{w \in \mathcal{W}'} |w|$. □

Although we have not observed this in practice, the quadratic term above (the sum of the squares of the hot pixel complexities) can be quite large. Using a more elaborate insertion algorithm, which we have not implemented, it is possible to reduce the total insertion cost to

$$O(n \log n + A + \sum_{h \in \mathcal{H}} |h| \log |h| + C). \tag{3}$$

To do so we need to exploit the fact that all ursegments associated with a particular fragment f possess a linear ordering within the region defined by the Minkowski sum of the fragment and a hot pixel, minus the two hot pixels at which f terminates — the ursegments neither terminate nor cross in that region. All these orderings can be computed and maintained in $O(\sum_{h \in \mathcal{H}} |h| \log |h|)$ time. During the VCD traversal associated with the insertion of a new ursegment t, every time t crosses a fragment f in the VCD outside a hot pixel, we can traverse the list of ursegments of f in the appropriate order, and look for intersections with ursegments, but only *within* the above Minkowski sum. We stop as soon as we detect a non-intersection, or enter one of the hot pixels at the ends of f, or t ends. The cost of the operations that do not yield a vertex of the ideal arrangement can be changed to either the hot pixel entered by t, or to an endpoint of t. It would also be highly desirable to eliminate the term C in the above bound, which is due to the search for the existing hot pixels entered by t (some, but not all such, will be found by the new modified VCD search explained here). Unfortunately, points can be within the same pixel, yet belong to trapezoids that are far away in the graph structure the VCD represents (think of a fan of closely spaced fragments going through the same pixel). In fact, in the worst case, C can be as large as $\Theta(n^2)$. Eliminating the cost of the C term will require us either to exploit the randomized hierarchy better, or to build and maintain some independent data structures.

We omit a detailed analysis of the deletion procedure. We only remark that we can implement the test for whether a vertex of a polysegment σ becomes deletable (after the deletion of the corresponding ursegment s) in time proportional to the logarithm of the number of ursegments crossing the hot pixel of that vertex.

8. Implementation and Verification

We have implemented the algorithms and data structures described above in

Allegro Common Lisp on a Silicon Graphics Indigo Elan. We began by constructing a naive and inefficient implementation of snap rounding. Each component of the more sophisticated implementation, which takes advantage of the algorithms described earlier, was developed and tested independently by replacing the corresponding component in the naive implementation. To verify the new "partially sophisticated" implementation, we generated large numbers of test cases and, using software to test whether two VCDs were identical, compared the output of the new implementation to that of the naive one.

For example, to implement the search for existing ursegments that pass through the new hot pixels, we used exhaustive versions of the search for new hot pixels and of the search for the existing hot pixels through which the new ursegment passes. We implemented the more efficient local search only for the existing ursegments that pass through the new hot pixels discovered by the exhaustive search.

This process quickly revealed errors in earlier versions of our algorithms. Usually the errors arose because we did not anticipate certain configurations of ursegments that have a very low probability of occurring. Only after many random trials were the errors exposed. To understand how these configurations caused our algorithm to fail, we wrote visualization tools tailored to our data structures. Using these tools, we could explore the problematic configurations in detail, and in most cases it rapidly became obvious why the algorithm failed – although not always how to fix it!

A number of factors were key to our implementation and verification of the software. The existence of a naive implementation simple enough for us to be confident that it was producing correct results provided a benchmark against which more sophisticated implementations could be tested. Our strategy of constructing the more sophisticated implementation by incrementally replacing components of the naive implementation made it much easier to identify the errors in our algorithms. The software that automated comparing one VCD to another made it possible for us to test the incremental algorithms with very large numbers of cases by comparing the resulting VCDs to those constructed by the naive algorithm. Finally, our tools to visualize and inspect local structures in the VCD helped us to better understand the algorithms, especially when they were producing incorrect results.

Figure 10 shows an arrangement of 50 ursegments and corresponding snap-rounded arrangement of fragments. The ursegments are on the top, and the fragments are on the bottom.

9. Future Directions

Our work leaves open several questions on the proper implementation of dynamic snap rounding. The only data structure currently used is the rounded VCD, with references from the fragments to the corresponding ursegments. As our discussion in Section 5 showed, we have difficulty in locating existing hot pixels that a new ursegment passes through. This is because the area of the plane covered by hot pixels is not explicitly represented in the VCD in any way, and thus a point in a hot pixel can be topologically arbitrarily far in the VCD from that pixel's center (which

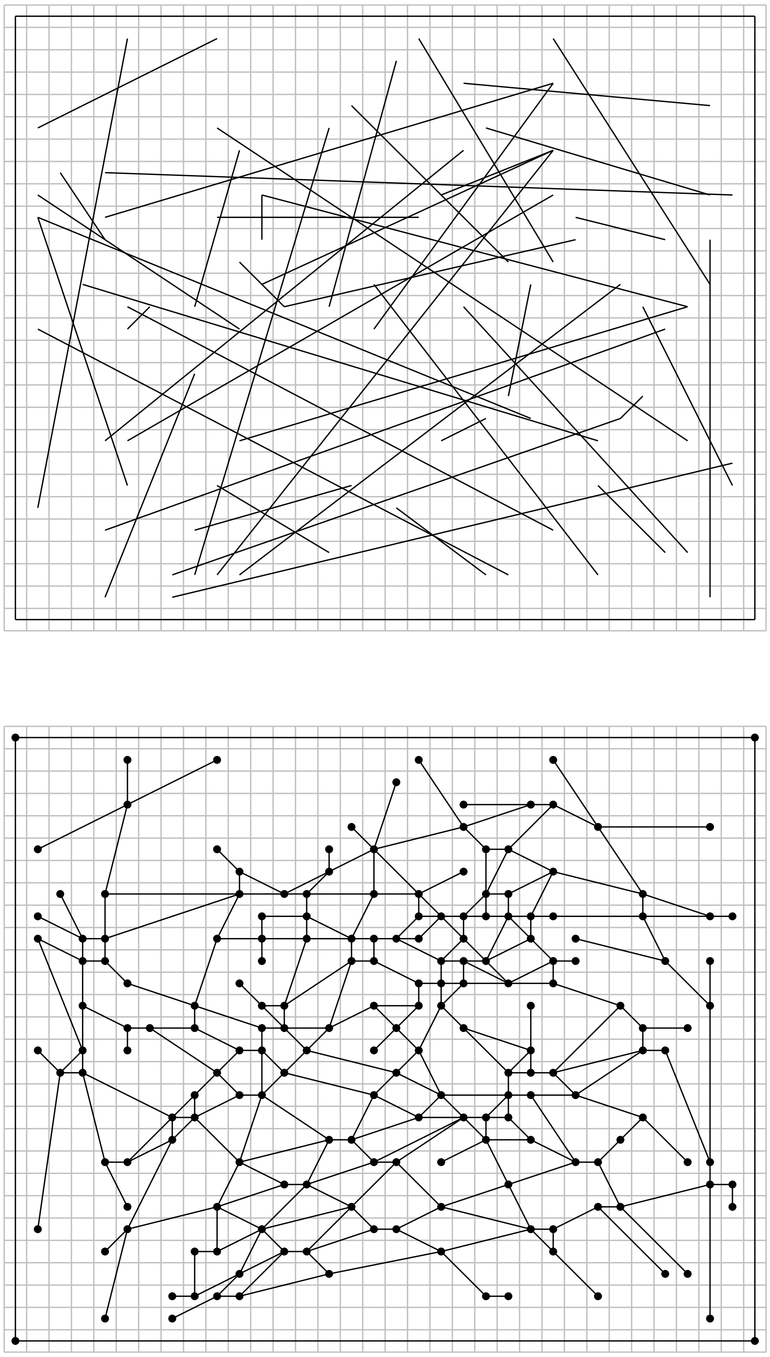

Figure 10: Top, an arrangement of 50 ursegments; bottom, the snap-rounded form of the arrangement.

is in the VCD). It seems worthwhile to investigate a data structure that would represent the hot pixels better and allow more efficient detection and traversal of hot pixel regions by a new ursegment.

This might also allow us also to remove the dependence on A, the complexity of the ideal arrangement, in the running time of our algorithm. In effect now we are paying to discover certain aspects of the micro-structure of the arrangement $\mathcal{A}$ inside the hot pixel regions, even though we really do not need to. Thus we could hope for an algorithm which is output sensitive in terms of the complexity R of the rounded arrangement only.

An extension of our work to handle arrangements of arcs of curves in the plane, or of planar or curved surface patches in space would also be highly desirable.

10. Conclusions

Trapezoidal decompositions, flat or hierarchical, are now part of the standard machinery of cuttings in Computational Geometry. Yet when we try to perturb and round these structures so as to make all their features exactly representable in finite arithmetic, many subtle issues arise. This is especially so in a dynamic context, as many of the invariants that guarantee the correctness and good performance of the infinite precision algorithms do not hold when we try to emulate these algorithms using the finite-precision structures we can actually store.

The key contribution of our paper has been to show how to marry the planar VCD rounding with dynamic operations on the arrangement of line segments it represents. We do so while maintaining as much as possible the performance of the ideal algorithms and producing in all cases a canonical rounding — the structure computed depends only on the set of segments present and not the history of the modifications made. In developing these techniques we saw a true interplay between theory and implementation. We have been able to employ simpler algorithms for operating on the rounded VCDs because we could go back and theoretically derive additional properties of snap rounding which then enabled us to prove the correctness of these simpler methods, as well as to analyze their performance.

We expect that this combination of theory, analysis, and implementation should apply to many other finite-precision models of geometric data structures and algorithms. We also believe that this type of algorithm will necessitate a new kind of analysis that elucidates the interaction between metric (distance, area) and combinatorial measures on geometric configurations.

Acknowledgments

We thank Dan Greene for useful discussions. Leonidas Guibas wishes to acknowledge support by NSF grants CCR-9215219 and IRI-9306544.

An earlier version of this paper appeared in the ACM Symposium on Computational Geometry, 1995.[17]

References

1. Daniel H. Greene, "Integer Line Segment Intersection," unpublished manuscript.
2. John D. Hobby, "Practical Segment Intersection with Finite Precision Output," submitted for publication.
3. Ketan Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms,* (Prentice Hall, Englewood Cliffs, NJ, 1994).
4. J. L. Bentley and T. A. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Trans. Comput.* **C-28** (1979) 643–647.
5. B. Chazelle and H. Edelsbrunner, "An optimal algorithm for intersecting line segments in the plane," *J. Assoc. Comput. Mach.* **39** (1992) 1–54.
6. K. Clarkson and P. Shor, "Applications of random sampling in computational geometry II", *Discrete Comput. Geom.* **4** (1989) 387-421.
7. Ketan Mulmuley, "A fast planar partition algorithm I", *J. Symbolic Comput.* **10** (1990) 253-280.
8. C. Hoffman, J. Hopcroft, and M. Karasick, "Towards implementing robust geometric computation," *Proc. Fourth Ann. Symp. on Computational Geometry,* 1988, pp. 106–117.
9. V. Milenkovic, *Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic,* Ph.D. Thesis, Carnegie-Mellon, 1988. Also Technical Report CMU-CS-88-168, Carnegie Mellon University, 1988.
10. K. Sugihara and M. Iri, *Geometric algorithms in finite-precision arithmetic,* Research Memorandum RMI 88-10, University of Tokyo, September 1988.
11. S. Fortune, "Stable maintenance of point-set triangulation in two dimensions," unpublished manuscript, AT&T Bell Laboratories. (An abbreviated version appeared in *Proc. 30th Ann. Symp. on Foundations of Computer Science*, 1989, pp. 494-499.)
12. L. Guibas, D. Salesin, and J. Stolfi, "Epsilon Geometry: building robust algorithms from imprecise calculations," *Proc. Fifth Ann. Symp. on Computational Geometry,* 1989, pp. 208–217.
13. S. Fortune and V. Milenkovic, "Numerical stability of algorithms for line arrangements," *Proc. Seventh Ann. Symp. on Computational Geometry,* 1991, pp. 334-341.
14. Daniel H. Greene and Frances F. Yao, "Finite-Resolution Computational Geometry," *Proc. 27th Ann. Symp. on Foundations of Computer Science,* 1986, pp. 143-152.
15. V. Milenkovic, "Practical methods for set operations on polygons using exact arithmetic," Proceedings of the 7th Canadian Computational Geometry Conference, 1995, pp. 55–60.
16. Michael T. Goodrich, Leonidas J. Guibas, John Hershberger, and Paul J. Tanenbaum, "Snap Rounding Line Segments Efficiently in Two and Three Dimensions," *Proc. Thirteenth Ann. Symp. on Computational Geometry,* 1997 (to appear).
17. Leonidas J. Guibas and David H. Marimont, "Rounding Arrangements Dynamically," *Proc. Eleventh Ann. Symp. on Computational Geometry,* 1995, pp. 190-199.