Snap Rounding Line Segments Efficiently in Two and Three Dimensions

Michael T. Goodrich[*]
Cent. for Geometric Computing
Johns Hopkins University
Baltimore, MD 21218
goodrich(jhu.edu

Leonidas J. Guibas[†]
Dept. of Computer Science
Stanford University
Stanford, CA 94305
guibas@cs.stanford.edu

John Hershberger
Mentor Graphics Corp.
1001 Ridder Park Drive
San Jose, CA 95131
john_hershberger@mentorg.com

Paul J. Tanenbaum
U.S. Army Research Lab.
ATTN: AMSRL-SL-BV
Ab. Prv. Gnd., MD 21005-5068
pjt@arl.mil

Abstract

We study the problem of robustly rounding a set S of n line segments in $\mathbf{R}^2$ using the *snap rounding* paradigm. In this paradigm each pixel containing an endpoint or intersection point is called "hot," and all segments intersecting a hot pixel are re-routed to pass through its center. We show that a snap-rounded approximation to the arrangement defined by S can be built in an output-sensitive fashion, and that this can be done *without* first determining all the intersecting pairs of segments in S. Specifically, we give a deterministic plane-sweep algorithm running in time $O(n \log n + \sum_{h \in H} |h| \log n)$, where H is the set of hot pixels and $|h|$ is the number of segments intersecting a hot pixel $h \in H$. We also give a simple randomized incremental construction whose expected running time matches that of our deterministic algorithm. The complexity of these algorithms is optimal up to polylogarithmic factors.

We also show how to extend the snap rounding paradigm to a collection S of line segments in $\mathbf{R}^3$ by defining hot voxels in terms of "close encounters" between segments in S, and we give an output-sensitive (though probably sub-optimal) method for finding all close encounters determined by the segments in S.

Key words. Robustness, finite precision, geometric rounding, line segments, arrangements.

[*]This research is supported by NSF grant CCR-9625289 and by U.S. ARO grant DAAH04-96-1-0013.

[†]This research is supported by NSF grant CCR-9623851 and US Army MURI grant 5-23542-A.

1 Introduction

Geometric objects typically live in a continuous geometric space such as $\mathbf{R}^2$ or $\mathbf{R}^3$. Yet computer representations of such objects are necessarily discrete, both because of the digital nature of computers themselves, and of the raster nature of the computer displays currently in use. Thus the need arises to represent geometric objects in a finite, prespecified resolution. Even if variable resolution is allowed, repeated geometric operations can rapidly increase the required precision. These considerations give rise to the fundamental problem of *rounding* geometric objects to a prespecified resolution. Though individually rounding the numerical attributes of the representation of a geometric object is usually straightforward, such rounding may violate various structural properties which the geometric object is supposed to satisfy, such as convexity, simplicity, etc. In this paper we are interested in rounding collections of line segments so as to guarantee certain topological consistency properties between the ideal collection and its rounded counterpart. Failure to guarantee this consistency can lead to erroneous results and algorithmic failures in further processing. We are interested in roundings that perturb and fragment the original input as little as possible, and which can be computed efficiently.

1.1 Some Approaches to Rounding

The problem of dealing with finite precision and robustness in geometric algorithms is fundamental, and there has been considerable work done on developing good approaches to this problem (e.g., see [2, 8, 9, 10, 11, 13, 15, 16, 17, 19, 22, 23]). Of particular relevance to this paper is the previous work done on producing rounded versions of arrangements of line segments. At a high level, of course, the goal of such a method is to round the given set of line segments so that each rounded version of a segment is "close" to the original segment and the important topological properties of the original configuration are preserved as much as possible. It is of prime importance in performing such a computation that the rounding be done efficiently, both in terms of the combinatorial size of the representation and in terms of the running time of the algorithm that performs this rounding.

Greene and Yao [10] introduced the framework of rounding line segments to a pixel grid. They gave a method that preserves the topology of a segment arrangement sufficiently, but at the expense of converting each individual line segment into a polygonal chain containing many subsegments. In addition, the running time of their method depends on both the combinatorial complexity of the output (the number of subsegments needed to represent rounded segments) and also on the number of actual intersections among the original segments. Subsequent to this early work on segment rounding, there have been several papers that have examined the arithmetic complexity (in terms of bits of accuracy) needed to construct arrangements [9, 16, 19]. The general framework of these approaches still involves the computation of all segment intersections, although possibly at a reduced bit complexity than a naive method might use.

One approach to the segment rounding problem that has been shown to be very promising, from the standpoint of the combinatorial complexity of the rounded representation, is the *snap rounding* paradigm introduced by Greene and Hobby [15] and studied in more detail by Guibas and Marimont [11]. Given a set S of n line segments in the plane and a regular pixel grid $\mathcal{G}$, this approach involves defining pixels in $\mathcal{G}$ as being "hot" if they contain segment endpoints or segment intersection points (the point features of the arrangement). Guibas and Marimont show that if one "snap rounds" each segment passing through a hot pixel to the center of that pixel, then one is guaranteed not to introduce any new crossings (although one may, of course, introduce new incidences). (See Figure 1.) This has the benefit of being combinatorially efficient, generating a minimal fragmentation of the input segments consistent with topological consistency. In addition, the size of the rounded arrangement will be almost always smaller than that of the ideal arrangement (though counterexamples are possible), and much more so with coarse pixel sizes.

In addition, Guibas and Marimont give an algorithm for constructing such a snap-rounded arrangement of S, including its vertical decomposition (i.e., its trapezoidal decomposition), in expected time $O(n \log n + A + \sum_{h\in H} |h| \log |h| + \sum_{w\in W} |w|)$, where A is the total number of pairs of intersecting segments in S, H is the set of hot pixels, $|h|$ denotes the number segments intersecting a hot pixel $h \in H$, W is the set of all pixels containing a vertical attachment (defined in the vertical decomposition), and $|w|$ is the number of segments intersecting a pixel $w \in W$. Their algorithm is dynamic, allowing also for efficient segment insertions and deletions. Alternatively, Hobby [15] describes a deterministic batch algorithm that first constructs the actual segment arrangement and then snap rounds it, resulting in an algorithm that runs in $O((n + A) \log n + \sum_{h\in H} |h|)$ time. Ideally,

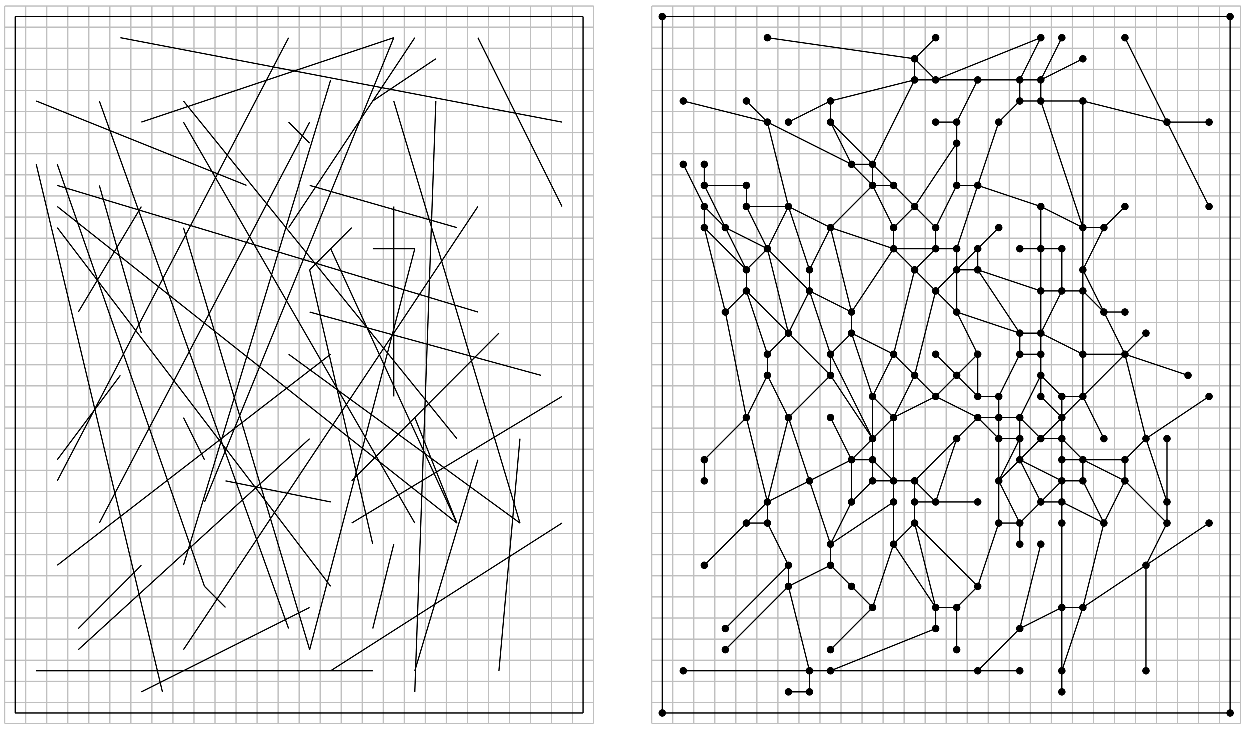

Figure 1: Left, an arrangement of 50 segments; right, the snap-rounded form of the arrangement.

however, one would desire an algorithm whose running time does not depend so heavily on A, reflecting the total number of intersecting pairs of segments in the ideal arrangement. In addition, one would like to eliminate any dependence upon other factors that do not contribute to the output size, such as the term $\sum_{w \in W} |w|$. A more desirable time bound would, for example, just depend upon n and the complexities of the hot pixels in H. This would give a method for rounding line segments that is efficient in terms of both its combinatorial and computational complexity.

1.2 Our Results

In this paper we give two output-sensitive algorithms for efficiently performing snap rounding of line segments in the plane. The first method is deterministic and runs in time $O(n \log n + \sum_{h \in H} |h| \log n)$. It is based upon a plane-sweep strategy; it avoids computing the full arrangement of S by erasing segments of S inside hot pixels. We also give a simple randomized incremental method whose expected running time matches that of our deterministic method.

In addition to providing efficient methods for 2-dimensional snap rounding, we also develop a framework for 3-dimensional snap rounding. In this context the n line segments in S are embedded in $\mathbf{R}^3$ and we wish to round them to a regular 3-dimensional voxel grid. Unlike the 2-dimensional case, however, these segments will typically not intersect at all. Thus, we characterize of the combinatorial complexity of a 3-dimensional set of segments in terms of n and the number of *close encounters* (or "near misses") between pairs of segments in S. We then propose a "hot" voxels definition in terms of those voxels containing segment endpoints and close encounters, and we propose a 3-dimensional rounded set of segments to be defined by taking all segments that intersect a hot voxel and snap-rounding them to the center of each such voxel. This retains the proximity of snap-rounded segments with original segments, but can be represented in the finite representation imposed by an integer-grid discretization of space. We also describe an output-sensitive method for finding all close encounters between the segments in S, which can then be used immediately to identify all the hot voxels determined by the segments in S.

We outline the main ideas behind our results in the sections that follow.

2 2-Dimensional Snap Rounding

Let S be a collection of n line segments in the plane, and let there be a regular grid $\mathcal{G}$ defining pixels in $\mathbf{R}^2$. In this section we describe two efficient methods for performing two-dimensional snap rounding of S, one deterministic and the other randomized. Both of these methods produce a vertical trapezoidal decomposition $\mathcal{S}$ of the snap-rounded arrangement of the segments in S, based upon the same strategy: first intersect segments with hot pixels and then collapse hot pixels to single points. This contrasts with the method of Guibas and Marimont [11], which is based upon a randomized incremental construction of $\mathcal{S}$ directly.

We say that a point in $\mathbf{R}^2$ is *critical* if it is the endpoint of a segment in S, or is the intersection point of two segments in S; we say that a pixel in $\mathcal{G}$ is *hot* if it contains a critical point. To *snap round* a segment s in S, then, we convert s into a polygonal chain (or *polyline*) that connects the centers of all the hot pixels that s intersects (in order of their intersection along s). Let H denote the set of all hot pixels, and let $\mathcal{H}$ denote the regions in $\mathbf{R}^2$ covered by hot pixels in H. In addition, let $\mathrm{int}(\mathcal{H})$ denote the interiors of all the hot pixels in H and let $\delta(\mathcal{H})$ denote the set of all line segments that bound hot pixels in H (so that, when viewed as sets of points, $\mathcal{H} = \mathrm{int}(\mathcal{H}) \cup \delta(\mathcal{H})$). For each segment s in S, define the *external fragments* of s to be the set of segments in $s \backslash \mathrm{int}(\mathcal{H})$. That is, the fragments of s are determined by "slicing" away all the portions of s that intersect hot pixel interiors. If a segment s in S has a non-empty fragment, then we form the polyline for s by moving the two endpoints of each fragment f of s to the centers of the two hot pixels that f is incident upon (if a fragment consists of just a single point, then we expand this to the segment joining the centers of the two abutting hot pixels upon which this point is simultaneously incident).

Let $\bar{S}$ denote the set of external fragments for the segments in S. The method we use to implement the above strategy is to construct a representation of $\mathcal{S}'$, the vertical decomposition of $\bar{S} \cup \delta(\mathcal{H})$ (using, say, the quad-edge data structure of Guibas and Stolfi [14]). We call $\mathcal{S}'$ the *pixel-clipped arrangement* of the segments in S. Note that none of segments in $\bar{S} \cup \delta(\mathcal{H})$ cross (although there will be intersections defined by fragment endpoints and hot pixel boundaries). Moreover, the combinatorial complexity of $\mathcal{S}'$ is proportional to the snap-rounded representation $\mathcal{S}$ of the arrangement of the segments in S. Since converting $\mathcal{S}'$ to $\mathcal{S}$ is a straightforward linear-time computation, we concentrate on the problem of constructing the pixel-clipped arrangement $\mathcal{S}'$ from the set of segments S.

2.1 Deterministic Snap Rounding

We present a deterministic plane-sweep method for constructing $\mathcal{S}'$, as described above. In what follows we refer to the original, unrounded segments as *ursegments*, to distinguish them from other segments used by the algorithm.

Our approach is based on the plane sweeping algorithm of Bentley and Ottmann [3] for constructing (unrounded) segment arrangements. We sweep over the ursegments from left to right, processing events, some of which are critical points. However, we dynamically modify the set of segments so that the sweepline processes only the leftmost critical point in each hot pixel. The sweepline is also used to detect all intersections between ursegments and hot pixel boundaries. The key idea is the following: when we detect a critical point in a pixel, we erase all the ursegment portions that intersect the (now known to be hot) pixel's interior. Simultaneously, we introduce the pixel's top/bottom boundaries into the sweep structure as horizontal line segments. This eliminates all ursegment intersections inside the pixel except the leftmost, and adds some ursegment/pixel boundary intersection events.

Our algorithm uses three data structures, which we describe in a syntax based loosely on C. First is a Bentley-Ottmann sweepline, which consists of a current x position `xpos`, a searchable list[1] storing the segments that intersect the vertical line $x =$ `xpos` in their y order, and an x-ordered priority queue of segment insertions, deletions, and intersections to the right of `xpos`—intersection events in the priority queue involve only adjacent segments on the sweepline [3, 4]. We denote the sweepline by `SL`. Second, each hot pixel `pix` with a critical point left of `xpos` has two searchable x-ordered lists of the ursegments that intersect its top and bottom left of `xpos`; these lists are `pix.toplist` and `pix.botlist`. Third is a searchable y-ordered list of the hot pixels that intersect $x =$ `xpos` and have a critical point left of `xpos`—call this list `Hcur`.

The output of the algorithm is an x-ordered list of hot pixels `H`, and for each hot pixel `pix` a set of ursegments `pix.segs` that intersect `pix`. The pseudocode below uses two subroutines: `pixel(Point p)` rounds point `p` to its containing pixel, and `heat(Pixel pix)` makes `pix` hot. In particular, `heat(pix)` finds all the ursegments that intersect `pix` left of the current `xpos`, erases the portion of these segments inside `pix`, initializes `pix.toplist` and `pix.botlist`, and introduces horizontal segments into `SL` at the top and bottom boundaries of `pix`. Whenever `heat(pix)` is called, there are no critical points between `xpos` and the left side of `pix`.

The algorithm processes five different kinds of events:

1. Ursegment endpoints. N.B. Right endpoints sort before left endpoints at the same x.

2. Ursegment intersections.

3. Ursegment/pixel boundary intersections.

4. Ursegment re-insertions. (These occur when an ursegment exits a hot pixel.)

5. Right ends of hot pixel boundaries. N.B. Type (5) events sort before type (4) events at the same x.

[1] A searchable list is any data structure that stores an ordered list and supports logarithmic-time insertions, deletions, and searches, and constant (amortized) time successor/predecessor queries. Balanced binary trees and skip lists are standard examples.

The algorithm begins by initializing the sweepline `SL` with events of types (1) and (2). No events of types (3), (4), and (5) exist yet. We process events in left-to-right order, as follows:

```
1.  Urseg_endpt(Point p, Urseg u)
    if (pixel(p) ∉ Hcur) heat(pixel(p)).
    if (p is the left end of u)
       Insert u into SL.
       Proceed as in case (3), beginning at (*).
    else
       Remove u from SL.
2.  Urseg_intersection(Point p)
    heat(pixel(p)).
3.  Urseg_pixbdy_intersection(Point p,
    Pixel pix, Urseg u)
    if (p is on the top of pix)
       append u to pix.toplist;
    else
       append u to pix.botlist.
(*) Add u to pix.segs, if not already there.
    Remove u from SL, as if u ended at p.
    if (u intersects the boundary of
       pix right of p at pp)
       Insert the event
       Urseg_reinsertion(pp, pix, u)
       into the priority queue.
4.  Urseg_reinsertion(Point p, Pixel pix,
    Urseg u)
    Insert u into SL at p.
    if (p is on the top of pix)
       append u to pix.toplist;
    else if (p is on the bottom of pix)
       append u to pix.botlist.
    Let ppix be the pixel adjacent to pix with
    p on their common boundary.
    if (ppix ∈ Hcur)
       Process an event
       Urseg_pixbdy_intersection(p, ppix, u).
       (A careful implementation—making pixel
       boundaries lie infinitesimally inside their
```

pixels—could avoid this test.)

5. `Pixbdy_end(Pixbdy bdy, Point p)`
   Remove `bdy` from `SL` at `p`.
   Append all pixels in `Hcur` to `H`, leaving `Hcur` empty.

The implementation of `pixel(Point p)` is trivial, so we focus on `heat(Pixel pix)`. We need the following lemma.

Lemma 1: *Given a pixel* `pix` $\notin$ `Hcur` *that intersects* $x =$ `xpos`, *let* `pixup` *and* `pixdown` *be the pixels in* `Hcur` *above and below it, if any. Let* `rect` *be the (possibly infinite) rectangle bounded by* `pixup`, `pixdown`, `xpos`, *and the* x *coordinate of the left side of* `pix`. *All the ursegments that intersect* `rect` *belong to* `pixup.botlist`, `pixdown.toplist`, *or the* y*-ordered list of segments on the sweepline* `SL`. *In each of these lists, the segments that intersect* `pix` *form a contiguous subsequence, which can be found in* $O(\log n + k)$ *time, where* k *is the length of the subsequence.*

Proof: By the definition of `Hcur`, no ursegment intersections or endpoints lie in `rect`. Any ursegment that intersects `pix` also intersects `rect`, and must intersect its top, bottom, or right side. Each of the three lists stores ursegments that intersect a particular line segment. If two ursegments s, s' in the list for a segment e intersect `pix`, then there is a line segment e' inside `pix` joining them. The four segments s, s', e, and e' form a quadrilateral. Any ursegment between s and s' in the list intersects e and enters the quadrilateral; it does not intersect s or s', so it must intersect e', and hence `pix`. We can find the subsequence of each list that intersects `pix` by searching in the list: for each segment s in the list that does not intersect `pix`, we can tell in constant time whether the portion of the list that intersects `pix` lies before or after s. □

We are now ready to present the implementation of `heat(pix)`.

```
heat(Pixel pix)
Insert pix into Hcur.
Find the ursegments that intersect
pix left of xpos, using Lemma 1.
For each ursegment u that intersects
pix left of xpos
   Add u to pix.segs.
   if (u intersects pix right of xpos)
      Remove u from SL.
      if (u intersects the boundary of pix
         right of xpos at a point p)
         Schedule an event
         Urseg_reinsertion(p, pix, u).
Build pix.toplist and pix.botlist
by sorting pix.segs.
Insert horizontal segments into SL at
the top and bottom boundaries of
pix, extending from xpos to the right
end of pix. For each pixel boundary
bdy inserted, with right endpoint p,
schedule an event Pixbdy_end(bdy,
p).
```

Theorem 2: *Given a set* S *of* n *line segments in the plane and a regular pixel grid* G, *one can snap-round the segments in* S *to* G *in time* $O(n \log n + \sum_{h \in H} |h| \log n)$, *where* H *is the set of hot pixels and* $|h|$ *is the number of segments intersecting a hot pixel* $h \in H$.

Proof: A pixel is hot iff it contains an ursegment endpoint or intersection. The algorithm detects these events in cases (1) and (2). Ursegments are erased from `SL` only inside pixels already known to be hot, so all hot pixels are detected.

Ursegment/hot pixel incidences are found in case (3) and in `heat(pix)` (called from cases (1) and (2)). This finds all the incidences, because any ursegment that intersects a pixel `pix` without starting or ending there must intersect either the top of `pix`, the bottom of `pix`, or all vertical segments connecting top and bottom. The subroutine `heat(pix)` finds all incidences with the top and bottom of `pix` left of `xpos`, as well as all incidences with the vertical segment spanning `pix` at `xpos`. Case (3) finds all incidences with top and bottom to the right of the `xpos` where `heat(pix)` was called.

Let $m = \sum_{h \in H} |h|$. It is not hard to see that the running time of `heat(pix)` is $O(k \log n)$, where k is the number of segments added to `pix.segs`. This sums to $O(m \log n)$ over all invocations of `heat()`. The rest of the algorithm requires only $O(\log n)$ time per event. The number of events of type (1) is $2n = O(m)$; the number of events of types (2) and (5) is $O(h)$, where h is the number of hot pixels, which is $O(m)$; and the number of events of types (3) and (4) is $O(m)$.

Once all the ursegment/hot pixel incidences are known, the order in which the segments intersect the pixel boundary can be obtained in $O(m \log n)$ time by sorting. (The order is already known for the intersections on the top and bottom of each hot pixel.) Given this information, it is straightforward to compute the snap-rounded arrangement $\mathcal{S}$. □

In the next subsection we describe a simple randomized algorithm whose expected running time matches this bound.

2.2 Randomized Snap Rounding in $\mathbf{R}^2$

In this section we give a randomized incremental construction (RIC) for building the pixel-clipped arrangement $\mathcal{S}'$ of the segments in S. The basic approach is similar to that of previous RIC's for constructing segment arrangements (e.g., see [5, 6, 18, 20, 21]). We again maintain a trapezoidal decomposition of $\mathcal{S}'$, except that here we dynamically "clip" the current subdivision each time we discover a new hot pixel.

In the usual RIC of line segment arrangements, two different operations need to be addressed. One is the *point location* of (typically the left) endpoint of a new segment s in the vertical trapezoidal decomposition of the arrangement of the segments inserted so far. The other is the *propagation* of s through the trapezoidal decomposition, in order to discover the intersections of s with existing segments, and to update the trapezoidal decomposition in the process. The point location step is normally handled by maintaining a conflict graph between (uninserted) segments and trapezoids [5], or by the "history-dag" technique of [12].

Like these methods, we also build the trapezoidal decomposition of the pixel-clipped arrangement $\mathcal{S}'$ in an incremental manner, by inserting all the segments in S into this arrangement one after the other, in a random sequence. But we differ from these schemes for ideal line segments in several important ways. Fist of all, because we are dealing with a fixed pixel grid, we can completely finesse the point location issue. We simply initialize our vertical decomposition of the pixel-clipped arrangement to be the vertical decomposition of all the hot pixels containing segment endpoints. We can easily compute this decomposition by a line sweep in time $O(n \log n)$. We also initialize each of these hot pixels with a dynamic "by-pass" structure that allows us to trace other segments though them efficiently. Secondly, during the propagation stage in the insertion of a new segment s, we may discover new intersections between s and other existing segments that lie outside of the currently known hot pixels. Whenever that happens, we need to create a new hot pixel h corresponding to the new intersection, clip out from $\mathcal{S}'$ the portion corresponding to h, and initialize a new by-pass structure for h containing all the current segments intersecting h. We illustrate the insertion of a random segment in $\mathcal{S}'$ in Figure 2.

Let us now be more precise about these operations; call s the next segment in S to be added. We locate the (existing) hot pixel h containing the left endpoint of s through a simple indexing operation. This can be done in time $O(\log n)$ by keeping on the side a binary tree of all the pixel grid columns containing hot pixels (due to endpoints) in the initial $\mathcal{S}'$, and for each node of that tree another binary tree of all the hot pixels in that column. In practice, of course, we would simply index into the pixel array using the endpoint coordinates. Starting now from where s exits the boundary of h, we trace s into the pixel-clipped decomposition $\mathcal{S}'$. The segment s may have to be propagated through a trapezoid of the decomposition or through an existing hot pixel, and in the process it may cross a vertical attachment, another segment, or a hot pixel boundary.

For every hot pixel h, we will maintain an ordered list of the intersections of the boundary of h by the current segment as a dynamic binary search tree (such as a red-black tree). By traversing this tree we are able to propagate a new segment, such as s, through h in $O(\log n)$ time;

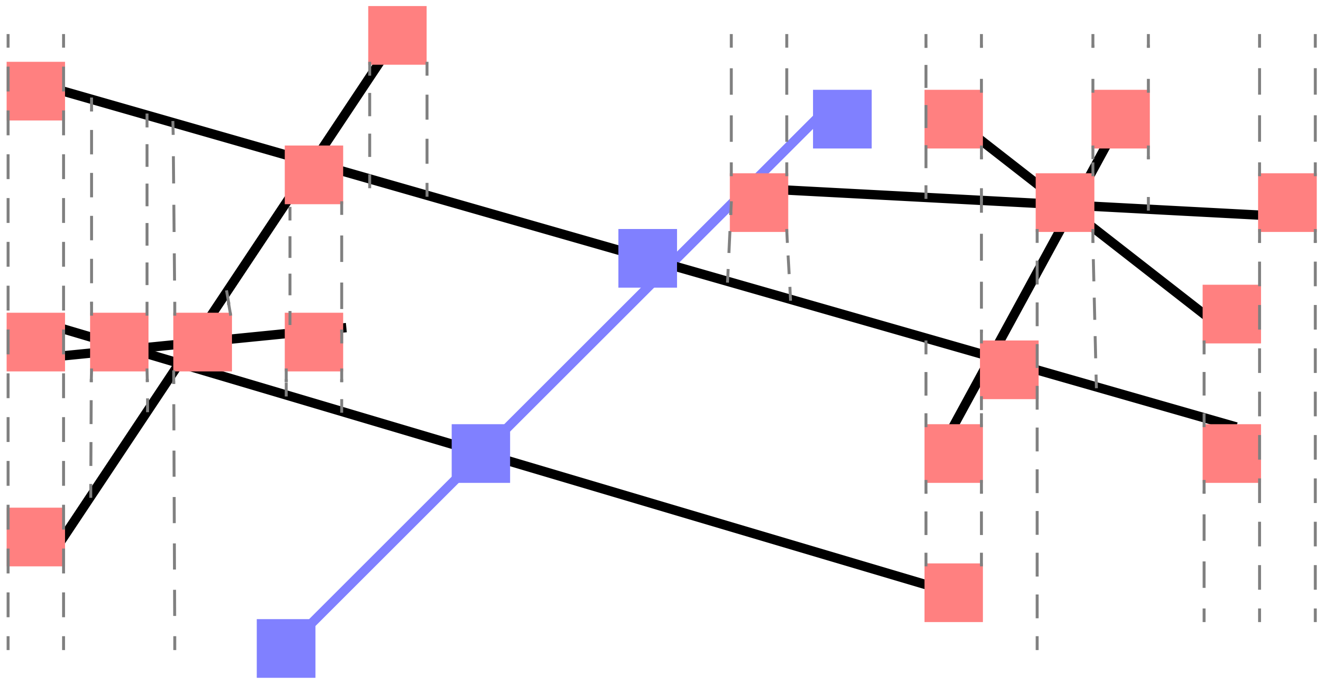

Figure 2: An illustration of the insertion of a random segment in $\mathcal{S}'$.

we can also update the representation to include s in $O(\log n)$ time. Thus the traversal by s of a standard trapezoid in $\mathcal{S}'$ takes as usual constant time, while the traversal of a hot pixel uses the above by-pass structure. If s exits a standard trapezoid by crossing a vertical attachment, then that attachment has to be shortened to end at s.

The most interesting situation arises when s exits a standard trapezoid by crossing an existing segment r at a point P. In this case we have discovered a new hot pixel h, and there is more work to do. In order to clip out from $\mathcal{S}'$ the part corresponding to h, we walk in $\mathcal{S}'$ from P to the (say) north-east corner of h (in a straight-line fashion), and then counterclockwise around the entire boundary of h. In this process we will discover all the current segments crossing h (each at most two or three times)[2]. Notice that in this traversal of $\mathcal{S}'$ we will not encounter any other hot pixels (obviously), and that the number of vertical attachments we can meet is at most four. Thus the work involved is proportional to the number of segments currently intersecting h. We can now excise h from $\mathcal{S}'$ and replace it by a by-pass structure reflecting the segment crossings with its boundary. We also add any necessary new vertical attachments emanating from the corners of h in $O(1)$ time.

This finishes the description of our RIC algorithm for the pixel-clipped arrangement. The analysis is very straightforward. Clearly the point-location cost is $O(\log n)$ per segment. Also, the cost of detecting the new hot pixels, of forming the hot pixel by-pass structures, and of tracing segments through them and then adding these segment to the by-pass structure is all accounted for by the term $O(\sum_{h \in H} |h| \log n))$. Note also that these are both worst-case bounds. The remaining cost of our algorithm is that of propagating segments through trapezoids by crossing vertical attachments. Each such step "clips," or shortens, such an attachment. Because the segments are added in a random sequence, any specific vertical attachment will be clipped in this way an expected $O(\log n)$ times (this is just the left-to-right minima in a random permutation problem). Note that a vertical attachment can also be clipped by the creation of a new hot pixel, but we can charge this clip-

[2]Some tiny segments may be entirely inside h, but these contract to a point after snap-rounding.

ping to the hot pixel responsible for it. This additional clipping can only help the expected $O(\log n)$ bound above. Thus we have:

Theorem 3: *Given a set S of n line segments in the plane and a regular pixel grid G, one can snap-round the segments in S to G in expected time $O(n \log n + \sum_{h\in H} |h| \log n)$ by a randomized incremental construction.*

3 3-Dimensional Snap Rounding

Let us now consider the problem of snap rounding a set S of n line segments in $\mathbf{R}^3$. In this case we assume a bounding box in $\mathbf{R}^3$ containing S has been partitioned into unit-cube voxels centered on the points with integer coordinates. We propse a notion of 3-dimensional snap rounding that is defined as follows. For distinct input segments r and s, consider the distance between them and define their *connector* to be the segment rs between their points of closest approach. We assume that no two input segments are parallel, so these points of closest approach are unique.

A connector will be called *short* if its length in the L_∞ metric is 1 or less. In this case, there is said to exist a *close encounter* between the two input segments. We define a voxel to be *hot* if it contains any endpoints either of input segments or of short connectors.

The rounding process proceeds analogously to the two-dimensional case described above. Every input segment s is transformed into a polyline σ such that the endpoints of σ are the centers of the hot voxels containing the corresponding endpoints of s and the bends in σ, corresponding to transits of s across hot voxels between its endpoints, are the centers of those hot voxels. Let us, then, address the computational issues involved with snap-rounding a set of n line segments in $\mathbf{R}^3$ using this notion of rounding.

3.1 Determining all close encounters

Suppose we are given a set S of n segments in $\mathbf{R}^3$. In this section we describe an efficient output-sensitive method for determining all pairs (s,t) of segments in S that are within an L_∞ distance of 1 from each other (which are all segments that determine a close encounter). From each segment s let us form the *tube* $\tau(s)$, which is the Minkowski sum of s with an axis-oriented unit cube centered at the origin. The following facts are immediate:

1. each tube $\tau(s)$ is a zonotope; it has at most twelve facets which are parallelograms;
2. segments s and t are within an L_∞ distance of 1 from each other if and only if their tubes $\tau(s)$ and $\tau(t)$ intersect.

We assume that no two segments are parallel; symbolic perturbation techniques can be used to guarantee that this is so [7]. It follows that two tubes $\tau(s)$ and $\tau(t)$ will intersect if and only if an edge of one of the tubes pierces (or touches) a face of the other. Therefore, let us form two sets from the collection of tubes for all the segments in S: the set E of all edges of the tubes, and the set F of all faces of the tubes. Each of these has clearly size $O(n)$. Note also that any intersection or contact between an edge e in E and a parallelogram f in F implies an intersection or contact between two tubes (except if e and f belong to the same tube), and any pairwise tube intersection will be captured this way.

We now use the range searching techniques for semi-algebraic varieties of Agarwal and Matoušek [1] to develop an efficient algorithm for reporting all the edge/face intersections. Consider a particular edge e and face f; orient the four edges of f consistently around f. Then the condition that e intersects f can be expressed by asserting that the line supporting e has positive orientation with respect to the four lines supporting the edges of f, and that that the plane supporting f separates the endpoints of v. By using the techniques of [1] we now preprocess all the faces in F so that, given a query edge e, we can quickly report all the faces that e intersects. This requires a six-level partition tree: four levels for the four sides of a face and two for the two endpoints of the query edge. Assuming that we want to use only linear space, the dominant query cost of this structure comes from the levels of the tree dealing with the line orientation conditions. By using Plücker coordinates, Agarwal and Matoušek [1] show how such a structure can be developed whose query time will be $O(n/s^{1/4} + k)$, where k is the number of reported faces intersecting e, using space and preprocessing of $O(s^{1+\delta}$, for any $\delta > 0$. By querying with all the edges in E and balancing the preprocessing and query costs, we can obtain all the edge/face intersections in total time $O(n^{8/5+\delta} + K)$ and space $O(n^{8/5+\delta})$, where here K denotes the number of intersecting tubes. Further details will be given in the full paper.

4 Conclusion

We have given output-sensitive methods for two-dimensional segment snap-rounding, in both deterministic and randomized settings. In both cases our methods have a running time that is sensitive to both the number of input line segments and also to the number of segments in a snap-rounded representation. We have also given an extension of the snap-rounding notion to three-dimensional segments and we have given an output-sensitive method for snap-rounding segments in $\mathbf{R}^3$ as well. We feel that an interesting direction for future work is to explore the variety of topological properties that are preserved by this notion of three-dimensional snap rounding.

Acknowledgement

We would like to thank Pankaj Agarwal for helpful discussions involving three-dimensional range searching.

References

[1] P. K. Agarwal and J. Matoušek. On range searching with semialgebraic sets. *Discrete Comput. Geom.*, 11:393–418, 1994.

[2] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluation of a new method to compute signs of determinants. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C16–C17, 1995.

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

[4] K. Q. Brown. Comments on "Algorithms for reporting and counting geometric intersections". *IEEE Trans. Comput.*, C-30:147–148, 1981.

[5] K. L. Clarkson. Randomized geometric algorithms. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 117–162. World Scientific, Singapore, 1992.

[6] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3(4):185–212, 1993.

[7] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.

[8] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations. *Internat. J. Comput. Geom. Appl.*, 5(1):193–213, 1995.

[9] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 334–341, 1991.

[10] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *Proc. 27th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 143–152, 1986.

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

[12] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[13] L. J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 208–217, 1989.

[14] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[15] J. Hobby. Practical segment intersection with finite precision output. Technical Report 93/2-27, Bell Laboratories (Lucent Technologies), 1993.

[16] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Towards implementing robust geometric computations. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 106–117, 1988.

[17] G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries in implicit Voronoi diagrams. Technical Report CS-96-16, Center for Geometric Computing, Comput. Sci. Dept., Brown Univ., Providence, RI, 1996.

[18] J. Matoušek and R. Seidel. A tail estimate for Mulmuley's segment intersection algorithm. In *19th International Colloquium on Automata, Languages, and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 427–438. Springer-Verlag, 1992.

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

[20] K. Mulmuley. A fast planar partition algorithm: part I. Technical Report 88-007, Dept. Comput. Sci., Univ. Chicago, Chicago, IL, 1988.

[21] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, volume 10 of *Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, 1993.

[22] K. Sugihara and M. Iri. Geometric algorithms in finite-precision arithmetic. Technical Report 88-10, Math. Eng. and Physics Dept., U. of Tokyo, Japan, Sept. 1988.

[23] K. Sugihara and M. Iri. Two design principles of geometric algorithms in finite-precision arithmetic. *Appl. Math. Lett.*, 2(2):203–206, 1989.