Geometric Streaming Algorithms with a Sorting Primitive (TR CS-2007-17)

Eric Y. Chen

School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1, Canada,
y28chen@cs.uwaterloo.ca

Abstract. We solve several fundamental geometric problems under a new streaming model recently proposed by Ruhl et al. [2, 12]. In this model, in one pass the input stream can be scanned to generate an output stream or be sorted based on a user-defined comparator; all intermediate streams must be of size $O(n)$. We obtain the following geometric results for any fixed constant $\epsilon > 0$:

- We can construct 2D convex hulls in $O(1)$ passes with $O(n^\epsilon)$ extra space.
- We can construct 3D convex hulls in $O(1)$ expected number of passes with $O(n^\epsilon)$ extra space.
- We can construct a triangulation of a simple polygon in $O(1)$ expected number of passes with $O(n^\epsilon)$ extra space, where n is the number of vertices on the polygon.
- We can report all k intersections of a set of 2D line segments in $O(1)$ passes with $O(n^\epsilon)$ extra space, if an intermediate stream of size $O(n + k)$ is allowed.

We also consider a weaker model, where we do not have the sorting primitive but are allowed to choose a scan direction for every scan pass. Here we can construct a 2D convex hull from an x-ordered point set in $O(1)$ passes with $O(n^\epsilon)$ extra space.

1 Introduction

Nowadays, applications with massive data sets are emerging rapidly in different areas, such as internet applications, geographic information systems, and sensor networks. Researchers have thus proposed algorithms that use small amounts of memory space under different space-conscious models. In in-place algorithms (e.g. see [5]), all input data are stored in memory, and the extra amount of memory used by the program is small or even constant. However, such algorithms are not suitable for data sets, which are larger than the size of memory. These massive data sets are considered under one-pass streaming models and multi-pass streaming models. In the one-pass streaming model, all input data are accessed once sequentially by the program, and the amount of memory used must be sublinear in the size of input or even constant. Algorithms under this model can process data sets larger than the size of the memory, but most of them can only compute approximate solutions. An alternative model is the multi-pass streaming model. In this model, the input data can be accessed sequentially multiple times. Algorithms under this model can compute exact solutions, but typically the number of passes taken by these algorithms grows when the size of input gets larger. In computational geometry, fundamental problems have been considered in all these models [5–7]. Not only efficient algorithms are proposed, but lower bounds [7] are also proved for several problems.

Ruhl et al. [2, 12] recently proposed a practical streaming model augmented with a sorting primitive, which will be defined precisely at the end of this section. Because sorting is a fully optimized operation under most systems, sorting a stream can be considered an atomic operation. In Ruhl et al.'s model, sorting the data stream once is counted as one pass on the data stream.

In this paper, we study several of the most fundamental problems in computational geometry [9–11] in the streaming model augmented with a sorting primitive. For any fixed ϵ, all of the following problems are solved with a constant number of passes and $O(n^\epsilon)$ space, where n is the size of input. These are the first results for these problems that achieve simultaneously a small number of passes and a reasonably small amount of space in a realistic streaming model. Specifically, these problems are: 1. constructing the convex hull from a set of 2D points, 2. constructing the convex hull from a set of 3D points, 3. constructing a triangulation of a simple polygon. Some of these problems have been even proved unsolvable with a constant number of passes in the original multi-pass streaming model [7]. If the size of the intermediate data stream is allowed to be larger, namely $O(n+k)$, we can solve a fourth problem: report all intersections of a set of 2D line segments with a constant number of passes and $O(n^\epsilon)$ space in memory. In a weaker model, which will also be defined precisely at the end of this section, we can still construct a 2D convex hull from an x-ordered point set in a constant number of passes with $O(n^\epsilon)$ space in memory. This has also been proved unsolvable in the multi-pass streaming model.

Note that, as Ruhl et al. [2, 12] showed, many parallel circuits, and consequently many parallel algorithms, can be simulated in the streaming model augmented with a sorting primitive. Several known geometric algorithms [1] can be simulated in this model directly. However, all of these transformed algorithms take $O(\operatorname{polylog} n)$ passes. Our algorithms are the first solutions only taking $O(1)$ passes.

Some of the techniques we use are standard. For example, for the 3D convex hull and segment intersection problems, we adapt standard random sampling techniques. To solve the triangulation problem, however, we need to introduce some new geometric observations and use a combination of several ideas.

1.1 The Streaming Model with a Sorting Primitive

In this subsection, we precisely define the streaming model with a sorting primitive (*stream-sort* model for short), describe one divide-and-conquer technique that we will use throughout in this paper, and also define a weaker model, which we call the *direction-flexible* streaming model.

In the stream-sort model, the input data are given in one data stream. There are two ways to access these data. One is the scan pass. The input stream is scanned sequentially, and one output stream is generated. The other is the sorting pass. The input stream is sorted based on a user-defined comparator, and data are sorted in the output stream based on that order. The generated output stream of data is called an *intermediate* stream. In the next scan, this intermediate stream becomes the input stream, and another output stream is generated. All intermediate streams must be of size $O(n)$. At the end, we count how many passes are used in total, and how much space is used in memory by the program in the scan passes.

Divide-and-conquer is a general strategy commonly used under this model. With this strategy, the data set of the problem is divided into data sets for several subproblems. We describe a technique that can help us solve all subproblems simultaneously in one pass.

This technique can be described as follows. Given a stream containing multiple independent data sets and one data set per subproblem, if the elements of each data set are grouped together in the stream, we can process these data sets one after another in a single pass. In the scan pass, after scanning over the data set for one subproblem, we reset memory for the data set of the next subproblem. Using a sorting pass, we can group the element for the same data set together. This can be done by adding a field in each element to identify which data set it belongs to. (The layout of the stream after the sorting pass is shown as fig. 1.) Storing this extra field only lengthens the stream by $O(n)$.

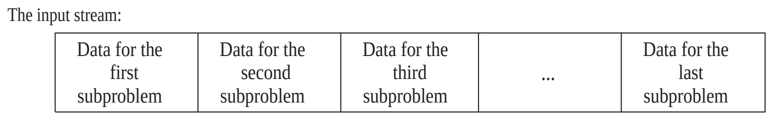
The input stream:
Data for the first subproblem
Data for the second subproblem
Data for the third subproblem
...
Data for the last subproblem

Fig. 1. The layout of the stream after a sorting pass

Now we treat the recursive calls in a divide-and-conquer algorithm as a tree structure. Each node represents one subproblem we need to solve. We can solve all problems in the same level of the recursion tree in the same pass. Therefore, we can bound the number of passes by the number of levels in the recursion tree.

Since the sorting pass is the most expensive part, we define a simpler and self-contained model in which the sorting pass is not allowed. Instead, we can only choose a direction to scan the stream (forward or backward). We call this model the *direction-flexible* model. This model is weaker than the stream-sort model. For example, it is impossible to sort data in order in a constant number of passes in the direction-flexible model, if the extra space allowed is sublinear.

2 2D Convex Hulls

In this section we give an algorithm that constructs the convex hull of a set of 2D points in $O(1)$ passes with $O(n^\epsilon)$ extra space under the stream-sort model. Constructing the convex hull of a set of 2D points in primal space is equivalent to constructing the lower envelope of a set of halfplanes in dual space [9]. In the pseudocode below, the input H is the set of halfplanes and the output L is the lower envelope of H. The parameter B in the pseudocode below will be determined later in this section.

```
Algorithm 2D_Envelope($H$)
    If $|H| \leq B$
        Solve the problem directly in memory and return $L$
    Divide $H$ in $B$ disjoint sets of equal size
    For each subset $H_i$
        $L_i$ = 2D_Envelopes($H_i$)
    Merge $B$ lower envelopes $L_i$ to obtain the lower envelope $L$ and return $L$
```

Merging B lower envelopes can be done in a constant number of passes in the stream-sort model by a sweep from left to right as follows. We maintain all $O(B)$ edges intersecting the sweepline and a list of edges appearing in the merged lower envelope. Whenever the sweepline touches an intersection of two edges in the merged lower envelope, we add the appearing edge to the list. We sort all edges by the x-coordinates of their left endpoints in a sorting pass. We only keep the B edges intersecting the sweepline in memory in the scan pass. The intersections between edges are computed in memory, so the space needed is $O(B)$.

We view the recursive calls of our algorithm as a tree structure. Applying the technique presented in section 1.1, we preform all merges at the same level of the recursion tree in one round. In a sorting pass, we use the group identifier of each element as the primary key and the left x-coordinates of edges in the same group as the secondary key. In the scan pass, we simply merge each group of lower envelopes, one by one.

By setting $B = O(n^\epsilon)$, we ensure that there are $O(1)$ levels of the recursion tree. Thus we have:

Theorem 1. *The convex hull of a set of 2D points can be constructed in $O(1)$ passes with $O(n^\epsilon)$ extra space, for any fixed $\epsilon > 0$.*

Remark: The above algorithm runs in $O(n^{1+\epsilon})$ time. We can reduce the time bound to $O(n \operatorname{polylog} n)$ time as follows. In the merging procedure, the intersections between the sweepline and envelopes are a set of points moving vertically with constant speeds. Merging B envelopes is equivalent to keeping track of the lowest point among the set of moving points. This can be maintained by a *kinetic heap* with $O(B)$ space in memory [4]. Whenever a new edge is touched by the sweepline, the velocity of the corresponding vertex is changed.

3 3D Convex Hulls

Given a set of 3D points, we show how to construct its 3D convex hull in $O(1)$ expected number of passes with $O(n^\epsilon)$ extra space in the stream-sort model. We follow a random sampling approach [8, 10].

We transform the problem into dual space. Each point in primal space maps to a halfspace in dual space. Constructing the convex hull of a set of 3D points is equivalent to constructing the lower envelope of a set of 3D halfspaces [9]. We first describe our algorithm in the traditional memory model, and then modify it to fit in the stream-sort model. The notations are defined as follows. The cell Δ_f defined by a triangle f refers to the vertial prism underneath f. The set of planes intersecting Δ_f is denoted by H_f. In the algorithm, the input H is a set of n 3D halfspaces, and the output E is the set of faces in the lower envelope. (See [10] for the definition of the *canonical triangulation.*)

```
Algorithm: 3D_Envelope($H$)
Initialize an empty set $R$
If $|H| \leq B$
    Solve the problem directly in memory and return the answer
Repeat
    Sample a random subset $R$ of size $B$ in $H$
    Find the lower envelope $E_R$ of $R$
    Build the canonical triangulation $T$ for $E_R$
Until $\sum_{f \in T} |H_f| = O(n)$ and $\max_{f \in T} |H_f| = O((n/B) \log B)$
For each $f \in T$
    $E_f$ = 3D_Envelope($H_f$)
Merge all the $E_f$'s to form $E$ and return $E$
```

The set R is a randomly selected subset. By the analysis from Clarkson and Shor [8] for randomly selected samples, it is known that the expected value of $\sum_{f \in T} |H_f|$ is $O(n)$. Therefore, we have $\sum_{f \in T} |H_f| \leq cn$ with probability greater than a constant, for a sufficiently large constant c. It is also known [10] that $\max_{f \in T} |H_f| \leq c'(n/B) \log B$ with probability greater than a constant, for a sufficiently large constant c'. Therefore, the expected number of iterations before both conditions satisfied is constant.

We modify this algorithm to fit in the stream-sort model. We set B to n^ϵ. Thus, R and E_R takes $O(n^\epsilon)$ space in memory. The operation for choosing the set R can be done in one scan pass. Constructing E_R can be done in memory using $O(n^\epsilon)$ extra space.

By keeping T in memory, verifying the conditions to terminate the loop can be done by one scan pass. One iteration of the first loop can be done in $O(1)$ passes with $O(n^\epsilon)$ extra space.

For the second loop structure, we proceed as follows. We create a copy of h for each cell Δ_f intersected by the halfspace h. We also attach a label to each copy to identify the corresponding cell. This operation can be done in one scan pass. In the following sorting pass, we use the attached label as the key. All halfspaces are grouped together in the data stream. With the technique introduced in section 1.1, all subproblems in the same level of the recursion tree are solved simultaneously in one scan passes.

Because the total number of intersections between halfspaces and cells is $O(n)$, the duplication of halfspaces only lengthens the size of the intermediate stream by a constant factor times. The size of any subproblem is $O(n^{1-\epsilon}\log n)$, since $B = n^\epsilon$. The number of levels of the recursion tree is constant, since the size of the intermediate stream increases by a constant factor every round. Therefore, all intermediate streams are of size $O(n)$.

Because the first loop in the above algorithm terminates in a constant expected number of iterations, this algorithm takes $O(1)$ expected number of passes in one round. Therefore, the expected total number of passes is also $O(1)$.

The merging step at the end of the algorithm can be done by a sorting pass and a scan pass. In the sorting pass, we use the planes that facets as the key and group all facets in the same plane together. In the scan pass, we merge these facets within the same plane.

Theorem 2. *Given a set of 3D points, its convex hull can be constructed in $O(1)$ expected number of passes with $O(n^\epsilon)$ extra space, for any fixed $\epsilon > 0$.*

Remark: The algorithm can also be derandomized in the same bounds. Instead of making the set R random, we can use a $(1/B)$-*net* to make the set, where $B = O(n^\epsilon)$. A streaming algorithm by Bagchi et al. [3] can deterministically compute this $(1/B)$-net in one pass with $O(\operatorname{polylog} n)$ space.

4 Triangulation of Simple Polygons

Triangulating a simple polygon reduces to triangulating a set of disjoint line segments. Given a set L of n disjoint line segments in a plane, we show how to construct a triangulation of L (covering the convex hull of the endpoints of L and not using extra vertices) in $O(1)$ expected number of passes with $O(n^\epsilon)$ extra space, for any fixed $\epsilon > 0$.

Before we describe our algorithm, we define some terms. By a *unimonotone* polygon, we mean an x-monotone polygon with one edge connecting its leftmost and rightmost vertex. We call this edge the *long* edge of the polygon. See fig. 2.

Our algorithm consists of four major phases. In section 4.1, we explain the construction of a trapezoidal decomposition of the line segments. In section 4.2, we describe the transformation of the trapezoidal decomposition to a decomposition of unimonotone polygons. In section 4.3, we describe the decomposition of each unimontone polygons to a set of *special* polygons. In section 4.4, we show

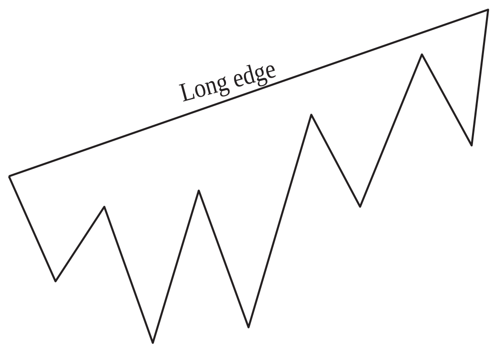

Fig. 2. A unimonotone polygon

how to triangulate a special polygon. In section 4.5, we put these four phases together to obtain the overall algorithm.

4.1 Trapezoidal Decomposition of Line Segments

We present a recursive algorithm constructing the trapezoidal decomposition of a set of disjoint line segments. The input L is a set of n disjoint line segments. The output T is the trapezoidal decomposition for T. We denote the set of line segments intersecting a trapezoid t by L_t. The parameter B will be determined later in this section. This algorithm uses a well-known random sampling approach [8, 10].

```
Algorithm: Trap_Decomp($L$)
If $|L| \leq B$
    Solve directly in memory
Repeat
    Randomly select a subset $R$ of size $B$ from $L$
    Build the trapezoidal decomposition $T_R$ of $R$
Until $\sum_{t \in T_R} |L_t| = O(n)$ and $\max_{t \in T_R} |L_t| = O((n/B) \log B)$
For each $t \in T_R$
    $T_t$ = Trap_Decomp($L_t$)
Merge all the $T_t$'s together to form $T$ and return $T$
```

The set R is a randomly selected subset. The analysis is similar to theorem 2. Therefore, we have $\sum_{t \in T_R} |L_t| \leq cn$ with probability greater than a constant, for a sufficiently large constant c, and $\max_{t \in T_R} |L_t| \leq c'(n/B) \log B$ with probability greater than a constant, for a sufficiently large constant c'. Therefore, the first loop iterates only a constant expected number of times.

In the stream-sort model, we keep the set R and T_R in memory. By setting $B = n^\epsilon$, these two structures only take $O(n^\epsilon)$ space in memory. With T_R in memory, we can check the conditions to terminate the first loop in $O(1)$ passes.

We use the same the duplication idea used in section 3. To prepare for the recursive calls in the second loop, in a scan pass, we create one copy of the segment for each trapezoid intersected and attach a label to each copy to identify the intersected trapezoid. In the sorting pass, we use the attached label as the key and group segments by the trapezoids intersected. Because $\max_{t \in T_R} |L_t| = O(n^{1-\epsilon}) \log n$, the number of levels of recursions is $O(1)$. Because $\sum_{t \in T_R} |L_t| = O(n)$, the intermediate stream for one level contains $O(n)$ line segments. Since there are only $O(1)$ levels of recursive calls, any intermediate stream contains $O(n)$ line segments.

The merging step can be done with one sorting pass and one scan pass. Each trapezoid is bounded by an upper edge, a lower edge, and two walls. In the sorting pass, we use the line segment of the upper edge as the primary key and the left-to-right order as the secondary key to sort all trapezoids. In the scan pass, trapezoids bounded by the same walls are all merged together.

We simultaneously solve subproblems in the same level in the same pass using the technique described in section 1.1. Our algorithm can build the trapezoid decomposition in $O(1)$ rounds. Since the expected number of pass to obtain a valid trapezoid is $O(1)$, our algorithm takes $O(1)$ expected number of passes in total.

Thus we have:

Lemma 1. *For any fixed $\epsilon > 0$, the trapezoidal decomposition of a set of disjoint n 2D line segments can be constructed in $O(1)$ expected number of passes with $O(n^\epsilon)$ extra space.*

4.2 Decomposition of Line Segments into Unimonotone Polygons

In this section, we show how to construct a decomposition of line segments into unimonotone polygons. This is a well-known algorithm described in [9]. We only adapt it in the stream-sort model. The input T is a set of trapezoids from the trapezoidal decomposition described in the previous subsection. The output M is the set of unimonotone polygons from this decomposition.

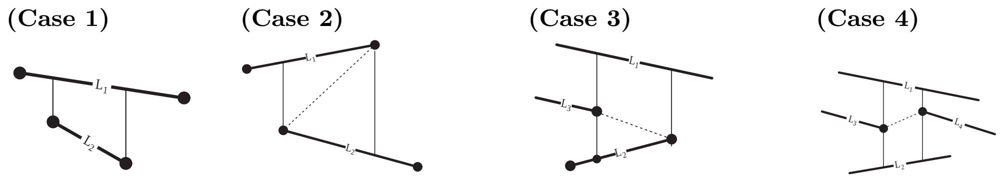

Fig. 3. Different cases to split a trapezoid

```
Algorithm: Monotone-Decomposition($T$)
Initialize $S = \emptyset$
/* step 1: split trapezoids */
Check for each $t \in T$
    /* These three cases are shown in fig. 3 */
    Case 1: the upper or lower edge of $t$ is a whole segment
        Put $t$ into $S$
    Case 2: two vertices are endpoints of line segments, but they are not from the same edge
        Draw the diagonal between the two vertices
        Split $t$ into $t_\uparrow$ and $t_\downarrow$ and put $t_\uparrow$ and $t_\downarrow$ into $S$
    Case 3: one vertex $p$ is an endpoint of a line segment
    and another endpoint $q$ of a line segment is on a vertical edge
        Draw the edge $\overline{pq}$
        Split $t$ into a triangle $\Delta_t$ and trapezoid $Q_t$ and put $\Delta_t$ and $Q_t$ into $S$
    Case 4: no vertices are endpoints of line segments
    and two endpoints, $p_1$ and $p_2$ are on the vertical edges
        Draw the edge $\overline{p_1 p_2}$
        Split $t$ into two trapezoids $T_1$ and $T_2$ and put $T_1$ and $T_2$ into $S$

    /*step 2: merge polygons in $S$ to unimonotone polygons */
    For each line segment $l$
        Sort all polygons using $l$ as the upper edge from left to right
        Merge all these polygons to form a unimonotone polygon $m$ and put $m$ into $M$
        Sort all polygons using $l$ as the lower edge from left to right
        Merge all these polygons to form a unimonotone polygon $m$ and put $m$ into $M$
Return $M$
```

Now we modify both steps of the algorithm for the stream-sort model. Step 1 can be simply done by a scan pass. Instead of writing the results to a data structure S, we write them into the output stream. For step 2, in a sorting pass, we use the segment of the upper edge as the primary key and the left-to-right order as the secondary key to group and sort polygons. In a scan pass, we merge the polygons, whose upper edges are of the same segment, to a unimonotone polygon and write the polygon to the output stream. We do the same for the polygons whose lower edges are of the same segment. Both of these two steps take $O(1)$ passes with $O(1)$ extra space.

4.3 Decomposition of a Unimonotone Polygon into Special Polygons

It is not obvious how to triangulate unimonotone polygons directly in the stream-sort model. In this section and the next, we introduce nontrivial new ideas that differ from the approaches in previous (sequential or parallel) polygon triangulation algorithms. The following definition is the key:

Definition 1. *Given a direction d, a unimonotone polygon is a* special *polygon at direction d, if its chain of edges is monotone in direction d and both vertices of the long edge are higher than any other vertices in the direction perpendicular to d.*

The decomposition is built recursively. Given the unimonotone polygon P, we divide the monotone chain into B parts with equal size, and build the upper hull for each part. Below the upper hulls, the decomposition will be built recursively. Above the upper hulls, we obtain a new unimonotone polygon, whose monotone chain Q is formed by at most B concave chains. See fig. 4. The following top-down sweeping algorithm decomposes Q into a set of special polygons. Interestingly, this part is inspired by a well-known algorithm for constructing the 2D maxima of a set of points [11]. The input Q is an x-monotone chain formed by B concave chains. The output C is the set of edges of the special polygons. Without loss of generality, we assume the right end is higher than the left end.

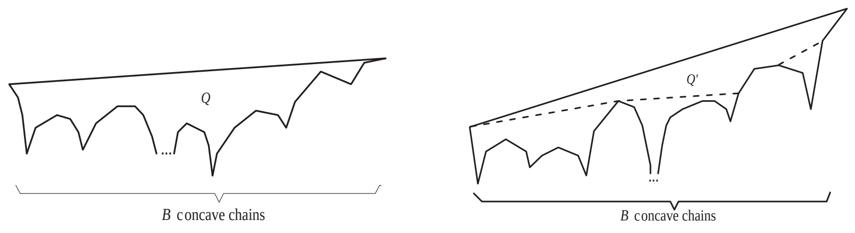

Fig. 4. Decomposition of Q into special polygons

```
Algorithm: Special_Decomp($Q$)
Initialize $v$ to the right end of the long edge and $C$ empty
Put all vertices along the $y$-direction in a sorted list $L$ in decreasing order
For each vertex $v_i$ in decreasing $y$-order
    If $v_i$ is left of $v$
        Add edge $(v_i, v)$ to $C$
        $v = v_i$
Return $C$
```

By this algorithm, we connect the right endpoint of the long edge to the left end with an xy-monotone chain, and all regions below this chain are special polygons along the x-direction, as one can easily see. Consider the region Q' above the xy-monotone chain. (See fig. 4.) This is also a special polygon, where, this time, the direction d is perpendicular to the long edge. In the scan pass, we only need to keep v and v_i in memory. Any created edge in C is written into the output stream as scan proceeds. With a sorting pass, we use sort all edges using the x-coordinate of the first point as the primary key and the x-coordinate of the second point as the secondary key. Then, with another scan pass, we can obtain the xy-monotone chain in order. Therefore, in the stream-sort model, the above algorithm takes $O(1)$ passes with $O(1)$ space in memory.

By setting B to n^ϵ, the number of levels of the recursion is $O(1)$. Then we simultaneously solve all problems in the same level of the recursion tree in one round. Because any edge we write to the output stream is an edge in the final triangulation and the size of the triangulation is $O(n)$, the size of all intermediate streams is $O(n)$. Since the upper hulls are computed by our algorithm in section 2, it takes $O(B)$ space in memory with $O(1)$ passes.

4.4 Triangulation of a Special Polygon

We build a triangulation for a special polygon, say in the direction of the x-axis, by a top-down sweeping procedure. In this algorithm, we maintain a tree structure *bridges* in memory. It stores a set of pairs *(left, right)*. Each pair corresponds to one portion of the chain intersecting the sweepline. They are ordered left to right in direction y. For p, which is a query point or a pair in *bridges*, its predecessor and successor in *bridges* refer to the bridge immediately left of and right of p, denoted as p^- and p^+, respectively. There are two types of events: 1. the sweepline touches a vertex which does not belong to an edge already intersecting the sweepline and 2. the sweepline touches a vertex which belongs to an edge already intersecting the sweepline. The input p is a special polygon whose monotone chain is monotone in x direction without loss of generality. For any other special polygon in other directions, we can rotate the coordinate plane, so that the polygon is a special polygon in direction of y-axis. The triangulated region constructed by the algorithm is illustrated in fig. 5.

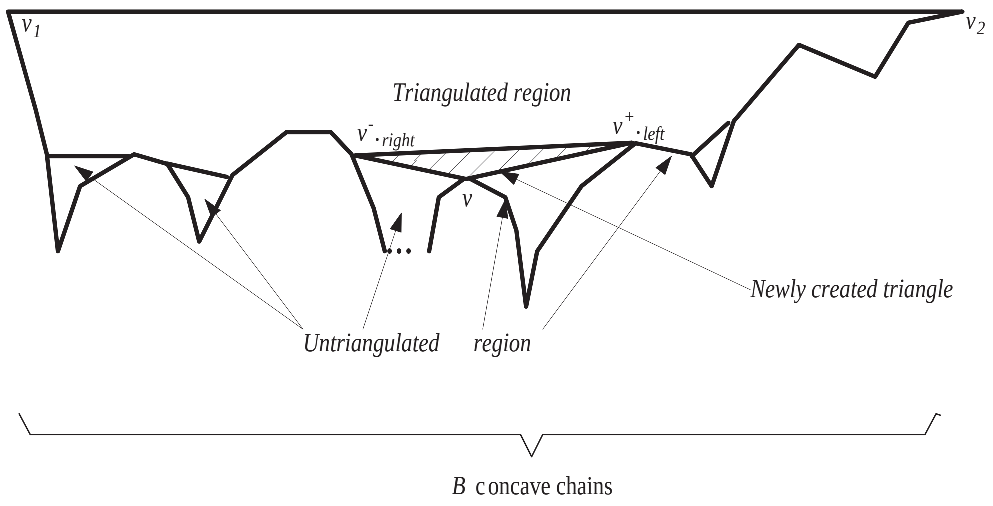

Fig. 5. Triangulating a special polygon

```
Algorithm: SpecialTriangulate($P$)
Put all vertices on the monotone chain top-down in direction $y$ in a sorted list $L$
Let the left end point of the vertex be $E_l$ and the right one be $E_r$
Initialize *bridges* with two pairs $(-\infty, E_l)$ and $(E_r, \infty)$
While $L$ is not empty
    Pick the next vertex $v$ from $L$
    Add triangle $(v^-.\text{right}, v, v^+.\text{left})$ to $T$
    Case 1: $v$ causes an event of the type 1 // see fig. 6 (right)
        Add $(v, v)$ to *bridges*
    Case 2: $v$ causes an event of the type 2 // see fig. 6 (left)
        If the left side of $v$ along the sweepline is inside $P$
            $v^+.\text{left} = v$
        Else if the right side of $v$ along the sweepline is inside $P$
            $v^-.\text{right} = v$
        Else
            Merge $v^-$ and $v^+$ to $(v^-.\text{left}, v^+.\text{right})$
    End of cases
End of loop
```

In the stream-sort model, we use a sorting pass to prepare L. Then we perform the loop part of the algorithm and write all triangles into the output stream in one scan pass, and store only *bridges* in memory. Therefore the extra space used in memory is linear to the maximum number of edges intersecting the sweepline.

Lemma 2. *Given a special polygon at direction d, it can be triangulated in $O(1)$ passes with $O(m)$ extra space, where m is the maximum number of edges of the polygon intersected by a line in direction d.*

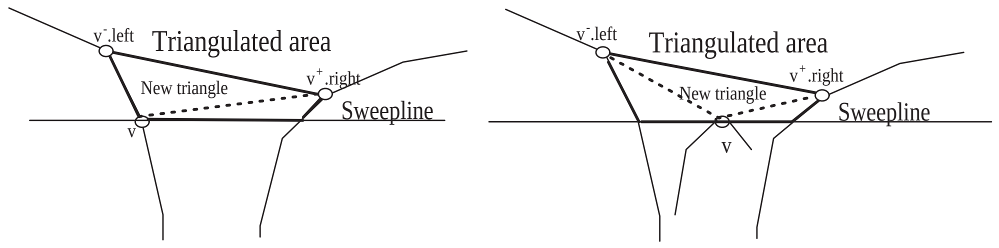

Fig. 6. Adding a new triangle into the triangulated area

Proof. We prove the following invariant: at any time, the portion of the polygon above the bridges is triangulated and lies above the sweepline. Initially, after the first two vertices v_1 and v_2 are read, the invariant holds, since v_1 and v_2 must be the vertices of the long edge by definition of special polygons, and so $\overline{v_1 v_2}$ lies above the sweepline. (See fig. 5.)

As the sweeping algorithm proceeds, the newly added vertices can be connected to the two adjacent vertices in the bridges without intersecting polygon edges (see fig. 6). Therefore, the invariant holds after the new vertex is added. $\square$

Remark: It is easy to find an example of a non-special unimonotone polygon where the above algorithm fails.

4.5 Triangulation of Line Segments

By solving multiple subproblems in one round, we simultaneously construct the triangulation of all special polygons decomposed from the same level of the recursive calls in the monotone decomposition algorithm described in section 4.3. Recall that the special polygons are decomposed from a unimonotone polygon whose monotone chain is composed of n^ϵ upper hulls. We conclude that the special polygon Q' above the xy-monotone chain and all obtained special polygons below the xy-monotone chain (see fig. 4) have $O(n^\epsilon)$ edges intersecting its sweepline. The extra space used to triangulate one special polygon is $O(n^\epsilon)$. Note that all of the four phases take $O(1)$ passes with $O(n^\epsilon)$ space. Thus, we have:

Theorem 3. *A triangulation of a set of disjoint line segments can be constructed in $O(1)$ expected number of passes with $O(n^\epsilon)$ extra space, for any fixed $\epsilon > 0$.*

Remark: The merging procedure used in sections 4.1 and 4.2 can be done in the same sorting pass, to minimize the number of expensive sorting passes in practice. This algorithm can also be derandomized by using the same $(1/B)$-net algorithm [3] mentioned in section 3.

5 Intersection Reporting for Line Segments

As another application of the trapezoidal decomposition algorithm, we can report all k intersections of a set of line segments with intersections in $O(1)$ expected number of passes with $O(n^\epsilon)$ extra space, if intermediate streams of size $O(n+k)$ are allowed, where k is the number of intersections.

We adapt the trapezoidal decomposition algorithm (section 4.1) to report intersections for a set of line segments with intersections. We modify the random sampling verification step used in the algorithm in section 4.1. Here we use $\sum_{t \in T_R} |L_t| \leq c(n + kB/n)$ [8], where k is the number of intersections detected so far. If this is a valid random sample, we build the trapezoidal decomposition on this sample, break the segment intersecting the boundary of the trapezoidal decomposition into two segments, one above the boundary and one below the boundary.

Corollary 1. *For any fixed constant $\epsilon > 0$, all intersections of a set of 2D line segments can be reported in $O(1)$ expected number of passes with $O(n^\epsilon)$ extra space, if size $O(n+k)$ intermediate streams are allowed, where k is the number of intersections between line segments.*

6 Constructing the 2D Convex Hull of x-Ordered Points in a Weaker Streaming Model

The sorting operation is expensive in the stream-sort model. We here describe how to construct the 2D convex hull of a set of x-ordered points in $O(1)$ passes with $O(n^\epsilon)$ extra space, in the direction-flexible model. This result also contrasts with the near-$\sqrt{n}$ lower bound result for the same problem given by Chan and Chen [7] in the original multi-pass stream model.

We only describe how to construct the upper hull, because constructing the lower hull is symmetric. Computing the lower hull simultaneously at most doubles the size of the intermediate stream.

Given a set P of 2D points and a vertical line h, we define the *bridge* at h as the edge of the upper hull of P intersecting h. To compute the bridge at h, we transform the problem in dual space. In dual space, each point becomes a halfplane and a bridge corresponds to an extreme point in the intersection of halfplanes. This point can be computed using linear programming. With $O(n^\epsilon)$ extra space in memory, this linear programming problem can be solved under the multi-pass model in $O(1)$ passes by the results of Chan and Chen [7].

We find all points not on the upper hull by divide-and-conquer. The input is a set P of 2D points sorted in x-order. In the output, all points not on the upper hull are marked. In the final pass, we scan this output, and report all unmarked points in x-order, which form the upper hull of P.

```
Algorithm: MarkUH($P$)
If $|P| = O(Bn^\epsilon)$
    Solve the problem directly in memory
Else
    Divide $P$ into $B$ groups: $P_1, P_2, ..., P_B$
    Find the set $H$ of vertical lines between any two adjacent groups
    For $P$, compute the bridges at each $h \in H$.
    Mark any $p \in P$ if $p$ is under one of these bridges
    For each $P_i$
```

```
MarkUH($P_i$)
```

Again, we solve all subproblems in one level simultaneously. For each subproblem, we need to compute $B-1$ bridges. The computation for each bridge corresponds one linear-programming problem. We use the multi-pass linear-programming algorithm mentioned above. This algorithm scans an array of data sequentially multiple times without modifying it, and the order of the scan does not affect the algorithm. However, between scans, addition information must be kept by the algorithm in memory.

In each subproblem, we compute all $B-1$ bridges by simulating this multi-pass linear programming algorithm. The memory contents of these simulations are kept in memory. After all data of one subproblem are scanned, we write all memory contents to the output stream after the data of that subproblem. Then the memory is reset for the next subproblem in the stream. Thus, in intermediate streams, data and memory contents of the multi-pass algorithm are interleaved. For the next scan, this output stream becomes the input stream. We start from the other end of the stream, so that we can reload the memory content of each subproblem back in memory, before any data point of that subproblem is accessed. After the final pass, all bridges are determined. They are written into the stream after the corresponding data set. To mark points under a bridge, we start from the other end. All bridges are loaded into memory, before the corresponding data set is accessed. This step takes $O(1)$ passes. By setting $B = O(n^\epsilon)$, the height of the tree is constant. Therefore, the total number of passes is $O(1)$ and the memory required is $O(n^{2\epsilon})$.

Besides one mark for each item, all extra information written into the stream consists of the memory contents produced by the multi-pass algorithm. At each level, the total number of bridges to compute is at most $O(\frac{n}{Bn^\epsilon})$. The total extra space is $O(n/B) = o(n)$.

By setting $2\epsilon = \delta$, we have:

Theorem 4. *Given a set of x-ordered 2D points, its convex hull can be constructed in $O(1)$ passes with $O(n^\delta)$ extra space in the direction-flexible streaming model, for any fixed $\delta > 0$.*

Acknowledgement

I thank Timothy Chan for suggesting the topic of geometric algorithms in the stream-sort model, and for help in the revision of the manuscript.

References

1. Aggarwal, A., Chazelle, B., Guibas, L., Ó'Dúnlaing, C., and Yap, C. Parallel computational geometry. *Algorithmica 3*, 1 (1988), 293 – 327.
2. Aggarwal, G., Datar, M., Rajagopalan, S., and Ruhl, M. On the streaming model augmented with a sorting primitive. In *Proc. of the 45th Annual IEEE Symposium on Foundations of Computer Science* (2004), pp. 540–549.
3. Bagchi, A., Chaudhary, A., Eppstein, D., and Goodrich, M. T. Deterministic sampling and range counting in geometric data streams. In *Proc. of 20th ACM Annual Symposium on Computational Geometry* (2004), pp. 144 – 151.
4. Brodal, G. S., and Jacob, R. Dynamic planar convex hull. In *Proc. the 43rd Symposium on Foundations of Computer Science* (2002), pp. 617–626.
5. Brönnimann, H., Chan, T. M., and Chen, E. Y. Towards in-place geometric algorithms and data structures. In *Proc. of 20th ACM Annual Symposium on Computational Geometry* (2004), pp. 239–246.

6. Chan, T. M. Faster core-set constructions and data-stream algorithms in fixed dimensions. *Computational Geometry: Theory and Applications 35* (2006), 20–35.
7. Chan, T. M., and Chen, E. Y. Multi-pass geometric algorithms. In *Proc. of 21st ACM Annual Symposium on Computational Geometry* (2005), pp. 180–189.
8. Clarkson, K. L., and Shor, P. W. Applications of random sampling in computational geometry, ii. *Discrete and Computational Geometry 4*, 1 (1989), 387 – 421.
9. de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry, Algorithms and Applications*. Springer, 1997.
10. Mulmuley, K. *Computational Geometry, An Introduction Through Randomized Algorithms*. Prentice Hall, 1994.
11. Preparata, F. P., and Shamos, M. I. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.
12. Ruhl, J. M. *Efficient Algorithms for New Computational Models*. PhD thesis, Massachusetts Institute of Technology, 2003.