Cartographic Line Simplification and Polygon CSG Formulæ in $O(n \log^* n)$ Time

John Hershberger[1]
Mentor Graphics
1001 Ridder Park Drive
San Jose, CA, USA 95131
john_hershberger@mentorg.com

Jack Snoeyink[2]
Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4
snoeyink@cs.ubc.ca

August 31, 1998

[1]Portions of this research were supported by DEC Systems Research Center.

[2]Supported in part by an NSERC Research Grant, the B.C. Advanced Systems Institute, and the IRIS NCE.

Running Title: Line Simplification and CSG Formulæ

Keywords: Dynamic convex hull, path hull, simple polygon

Correspondence Address: Jack Snoeyink
Department of Computer Science
University of British Columbia
201 - 2366 Main Mall
Vancouver, BC V6T 1Z4
Canada
fax : (604) 822-5485
email: snoeyink@cs.ubc.ca

Abstract

A constructive solid geometry (CSG) conversion for a polygon takes a list of vertices and produces a formula representing the polygon as an intersection and union of primitive halfspaces. The cartographers' favorite line simplification algorithm recursively selects from a list of data points those to be used to represent a linear feature, such as a coastline, on a map. By using a data structure that maintains convex hulls of polygonal lines under splits, both were known to have $O(n \log n)$ time solutions in the worst-case. This paper shows that both are easier than sorting by presenting an $O(n \log^* n)$ algorithm for maintaining convex hulls under splits at extreme points. It opens the question of whether there are practical, linear-time solutions to these problems.

1 Introduction: CSG formula computation and Line simplification

A plane polygon can be represented either by the sequence of vertices and edges around its boundary or by a boolean combination of "primitive" regions such as halfspaces. Solid modeling systems may convert from the former to the latter; this is a 2-dimensional example of a conversion from a *boundary representation (B-rep)* to a *constructive solid geometry (CSG) representation.*

Peterson [11] showed that a simple polygon always has a CSG formula using one primitive halfplane for each edge of the polygon. In fact, one can write down the formula by starting at the leftmost vertex and listing the halfspaces in the order that their edges appear around the polygon, inserting an "AND" for every convex corner and an "OR" for every reflex corner. The interesting part is to add parentheses appropriately.

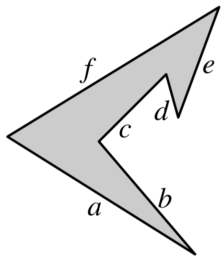

Figure 1: CSG formula is $a(b+((c+d)e))f$

In the dart in Figure 1, for example, the three terms a, $(b+((c+d)e))$, and f can be joined by "AND"s since extensions of segments a and f, and of the polyline $bcde$ by extending segments b and e, do not return to intersect the polygon. Segment e cannot appear at the top level because its extension intersects edges a and b. Dobkin et al. [1] gave an $O(n \log n)$ algorithm to recursively add parentheses. They maintain convex hulls of fragments of the polygon and split at hull vertices that are extreme in a direction determined by the directions of the first and last edges of each fragment.

To draw a coastline or other linear feature on a simple and readable map, one may need to perform *line simplification* to reduce the detailed data available in a database.

Cartographers have identified the recursive algorithm detailed in Douglas and Peucker [2] as best in mathematical [8] and perceptual [15] studies. This algorithm first approximates a polygonal line $p_1, p_2, \ldots, p_n$ with the segment $\overline{p_1 p_n}$. If the vertex $p_{\max}$ at maximum distance from the line $\overleftrightarrow{p_1 p_n}$ is within tolerance, this approximation is accepted, otherwise, the two polygonal lines from p_1 to $p_{\max}$ and from $p_{\max}$ to p_n are approximated recursively, as in Figure 2. This algorithm has been called Ramer's algorithm [13] in vision and the sandwich algorithm [14] in computational geometry.

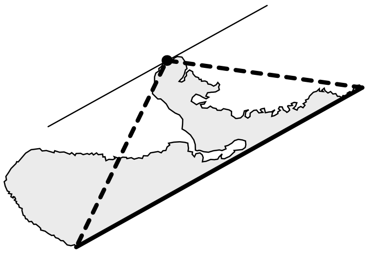

Figure 2: Line simplification

If implemented in a straightforward fashion, this algorithm has a worst-case running time of $\Theta(n^2)$ (and best-case of $\Theta(n \log n)$ when the tolerance is small). Because the vertices at maximum distance will be found on the convex hull, this same simplification can be computed in $O(n \log n)$ worst-case time [5, 6] using a convex hull data structure that supports splitting at extreme vertices. Here and throughout this paper, we assume that the polygonal line $p_1, p_2, \ldots, p_n$ is *simple*: the only intersections between its line segments occur where adjacent segments share a common endpoint. It should be noted that the algorithm does not guarantee simplicity of the output, however.

Thus, the problems of CSG formula computation and line simplification can both be solved by a data structure that stores fragments of a polygonal line and supports the operation of finding an extreme vertex in a particular direction and splitting the polygonal line there. The choice of direction depends upon the problem and the current fragment, so the extreme vertices are found on-line.

It is natural to ask whether these problems are easier than sorting. This paper gives an affirmative answer for CSG formula computation and for a modification of line simplification by presenting an $O(n \log^* n)$ algorithm for maintaining convex hulls of a polyline under splits. In Section 2 we review the "path hull" of Dobkin et al. [1] and other data structures for dynamic convex hulls. Section 3 describes our new data structure, which builds an augmented path hull data structure on "beads"—convex hulls of polylog-size fragments of the polygonal line. Section 4 analyzes the operations of finding extreme vertices and splitting for this structure.

Note: All logarithms in this paper are taken base 2. The iterated logarithm, $\log^* n$, is the number of times the logarithm function must be applied to reduce the argument to less than 2. It is a slowly growing function: $\log^* 16 = 3$, $\log^* 2^{16} = 4$, and the first n with $\log^* n \geq 5$ has about 20,000 decimal digits.

2 A brief review of 2-d dynamic convex hulls

As mentioned in the previous section, we wish to support splitting and finding extreme vertices of fragments of a polyline in the plane. At the cost of additional logarithmic factors, one could use general convex hull algorithms for points in the plane [12]. Overmars and van Leeuwen [10] showed that the divide-and-conquer algorithm can support finding extreme points in $O(\log n)$ time. Deletions and insertions of points take amortized $O(\log^2 n)$ time per operation; no algorithm is known to achieve amortized $O(\log n)$ time.

2.1 Hulls of polygonal lines

When the points lie on a simple polygonal line then the convex hull has additional structure, as many have observed. (See Figure 3.)

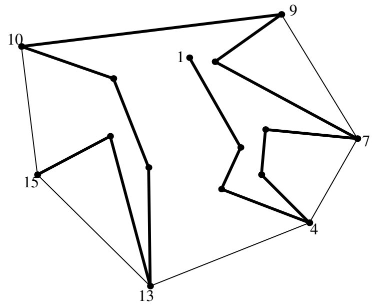

Figure 3: Vertex numbers on the hull

Observation 2.1 *If the vertices are numbered along the polyline, then the sequence of vertex numbers around the hull, when read counter-clockwise from the maximum, decreases to the minimum number and then increases to the maximum.*

Guibas et al. [3] used this observation to build an $O(n)$-size data structure for "subpath hull queries," which include the ability to find an extreme point of any contiguous fragment of the polyline in $O(\log n \log\log n)$ time after $O(n)$ preprocessing. This level of generality is not needed in the applications considered in this paper.

One consequence of Observation 2.1 is that common tangents to the convex hulls of two consecutive fragments of a simple polyline can be computed in polylogarithmic time. Guibas et al. [3] give a matching lower bound for simple array data structures (as well as an improvement using more complex data structures.)

Lemma 2.2 *Given array representations of the convex hulls of two consecutive fragments of a polygonal line with m vertices, one can compute the at most two tangents between them in $O(\log^2 m)$ time.*

Proof: Because the polyline fragments are consecutive, if, say, the first hull is contained inside the second, then it is contained entirely inside one of the two bays determined by the hull edges incident to the lowest numbered vertex.

For a specific example with hull A inside hull B, consider Figure 4. Vertex b on hull B has the lowest index, and is incident to hull edges e and e'; the bay defined by e contains the hull A. To detect this, one can test, in $O(\log m)$ time, whether the extreme vertices of A in the directions normal to e and e' are left of the lines through e and e'. If so, then A is inside.

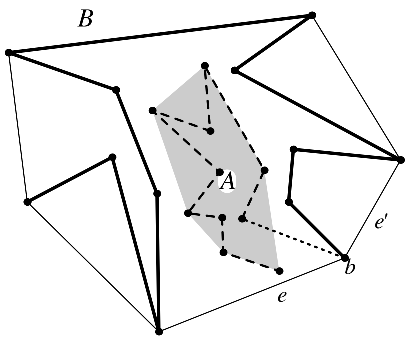

Figure 4: Hull A (shaded) in B

If neither one is inside the other, then we will have found a pair of "helper points"—a point on each hull that is not contained in the other. Guibas et al. [3] show how to use helper points to reduce to the problem of finding a common tangent of two intersecting, upper convex hulls. This can be solved by nested binary search: choose a median vertex on one hull and use binary search to determine the tangent to the other hull, if it exists. ■

2.2 Melkman's hull algorithm with a history stack

Melkman [9] uses Observation 2.1 in a different way to give an incremental algorithm that computes the convex hull of a polyline $p_1, p_2, \ldots, p_m$ in $O(m)$ time. It uses a doubly-ended queue (a deque) to store vertices of a convex hull in counter-clockwise order from front to back; the hull vertex with highest appears at both front and back. The deque can easily implemented as an array of size $2m$ with pointers to the front and back elements.

Start with the convex hull of the first two points by placing p_2, p_1, p_2 in the middle of the array. To add p_i, check if p_i appears on the convex hull; as noted in the proof of Lemma 2.2, it sufficient to inspect the edges incident on the hull vertex of highest index, which are found at the ends of the deque. If p_i does not appear on the hull, then no changes to the deque are required. Otherwise, p_i may hide some vertices of the previous hull; pop these hidden vertices from the front and/or back of the deque, and then push p_i onto both front and back to produce the convex hull of $p_1 \ldots p_i$. Since each vertex is pushed, and therefore popped, at most twice, the total time is $O(m)$.

Because this algorithm is incremental, it actually computes convex hulls of all prefixes of the polyline. Dobkin et al. [1] added a history stack that records the elements pushed onto and popped from the deque so that these prefix convex hulls can be recovered as vertices are deleted from the end of the polyline.

Lemma 2.3 *Given a polygonal line $p_1, p_2, \ldots, p_m$, where p_1 is the start or* anchor *vertex, one can build a convex hull data structure in $O(m)$ time that supports the operations of deletion from the (non-anchor) end in amortized $O(1)$ time and search for the extreme vertex p_d in a given direction in $O(\log \min\{d, m-d+1\})$ time.*

Proof: We sketch the analysis; greater detail on the structure can be found in Dobkin et al. [1].

Construction by Melkman's algorithm is described above; the history stack merely increases the constant. Deletion from the end is accomplished by playing back the history stack and reversing the operations; this can be charged against construction time so that deletion may be considered to take constant amortized time.

To enable the search for an extreme vertex, maintain a pointer to the hull vertex with lowest index. Note that the hull vertex with highest index appears at both the front and back of the deque. These lowest and highest indices split the deque into two arrays; one containing vertices with increasing numbers and the other with decreasing numbers. A constant-time computation on these vertices and their neighbors on the hull is sufficient to determine which array contains the extreme vertex. Search that array by starting two increasing-increment searches in parallel from the ends: Check the first, second, fourth, eighth, ..., from the end until the extreme vertex lies in the interval between the current vertex and the end, then finish with binary search on that interval.

When p_d is the extreme vertex desired, then there will be at most $2\min\{d, m-d+1\}$ vertices in the search interval because the vertices in the search interval appear in order along the hull.

Therefore, the search takes $O(\log \min\{d, m-d+1\})$ time. ■

We could say that Melkman-plus-history supports "one-sided" splits—splitting at a vertex produces a valid convex hull data structure for the first part of the polygonal line by simply deleting vertices from the second part. Of course, no structure is produced for the second part, so splits that occur near the beginning of the polyline waste most of the computation of Melkman's algorithm.

Dobkin et al. [1] used this observation to define a "path hull" data structure that supports two-sided splits in amortized $O(\log n)$ time. They choose an anchor in the middle of a polyline and use Melkman-plus-history to build two convex hull structures outward from the anchor. Thus, the splits that waste computation are those near the middle; since these splits now break the problem into two equal-sized subproblems, a credit scheme shows that the total splitting time is $O(n \log n)$. Extreme vertices are found in $O(\log n)$ time apiece by finding the two candidate extreme vertices, one on each of the convex hulls that make up the path hull, and returning the true extreme.

3 The bead hull data structure and its construction

To improve on the path hull structure sketched at the end of the previous section, we need to reuse more computation—we cannot afford to have all previous computation wasted by a single split. Less evident, but equally important, is the fact that the cost of finding an extreme vertex must be related to the size of the smaller polyline that will be created by splitting there—we cannot afford to find candidates at which we do not perform a split. (This will force a modification of the line simplification procedure in Section 5.)

We break a polyline of length n at vertices to form n/k fragments of length k, where k can be chosen to be $\log^2 n$. (We justify the choice of k in Section 4.) Adjacent fragments share a common endpoint. For each fragment, we build a convex hull using Melkman-plus-history as described in Lemma 2.3. We call such a fragment-with-hull a *bead*. We actually build the convex hull twice, once from each end. Because these data structures can be built in linear time (Lemma 2.3), we obtain

Corollary 3.1 *Beads of size k for a polyline of length n can be built in $O(n)$ total time and space.*

Our polyline will be subject to splits. Splitting at a vertex in a bead produces two *broken beads*—fragments of the polyline with length at most k that have Melkman-plus-history representations of their convex hulls built from the original endpoints towards the split. Further splitting a broken bead produces a smaller broken bead and an unstructured fragment between the two splits.

A *string of beads* is a two-level convex hull structure that represents the hull of a sequence of consecutive whole beads. The lower level consists of the convex hull arrays for the beads. The upper level is an array of tangents between beads, with pointers into the corresponding bead hull arrays.

Lemma 3.2 *Given m beads with size at most k, a string of beads can be built using a modification of Melkman's algorithm in $O(m \log^2 k)$ time.*

Proof: As in Melkman's algorithm for points, we can maintain the current list of tangents between beads in a deque with the tangents to the most recently added bead at the front and back of the deque.

To add the ith bead, we first check if it appears on the hull by testing against the common tangents to beads at the front and back of the deque. If so, we pop beads from the deque whose tangent lines intersect the bead—these will no longer be on the convex hull. We use the nested binary search of Lemma 2.2 to compute at most two tangents between the ith bead and the beads remaining at the front and back of the deque, then add these tangents and push the ith bead onto the front and back of the deque.

Because there are at most $2m$ tangents added, there are at most $2m$ tangents popped at a total cost of $O(m \log k)$. Adding new tangents costs $O(m \log^2 k)$ time. ∎

Our final data structure, the *bead hull*, consists of the following parts, which are depicted schematically in figure 5.

- An *anchor vertex* that is the common endpoint of two beads (broken or whole). An anchor is initially chosen in the middle of a sequence of beads.
- Two, possibly empty, strings of beads constructed to the left and right of the anchor.
- Two, possibly empty, broken beads; one at the end of each string.
- Tangents (at most four) from the broken beads to their adjacent strings.

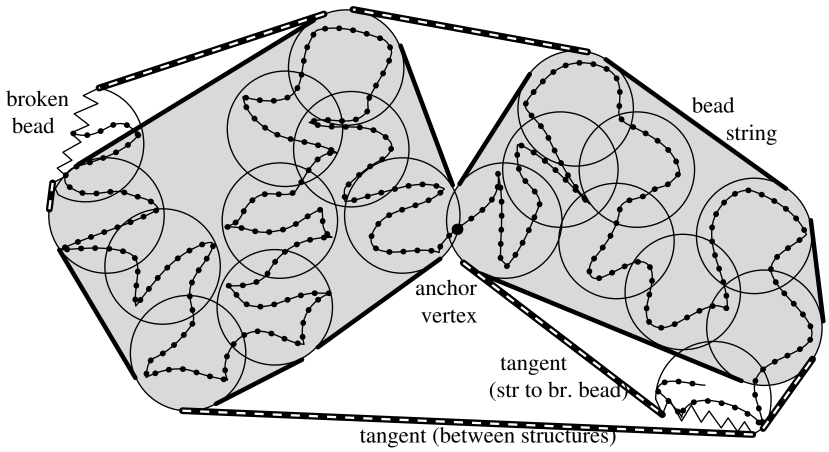

Figure 5: Schematic depiction of a bead hull with anchor in the middle

- Tangents (zero or two) between the structures to the right and left of the anchor.

Lemma 3.3 *Given $m > 1$ beads with size at most k, a bead hull can be built in $O(m \log^2 k)$ time.*

Proof: The anchor can be chosen in constant time and two strings of beads constructed in $O(m \log^2 k)$ time. By Lemma 2.2, the two tangents between the strings of beads can be found by nested binary search in $O((\log m + \log k)^2) \leq O(m \log^2 k)$ time. ∎

4 Analysis of bead hull operations

In a bead hull, finding an extreme vertex in a chosen direction is relatively easy because all hull edges are represented.

Lemma 4.1 *Given a bead hull representing $p_1, p_2, \ldots, p_n$, the extreme vertex p_d in a particular direction can be found in $O(\log \min\{d, n-d+1\})$ time.*

Proof: We can use increasing-increment searches in parallel from both ends, as in Lemma 2.3. The two-level structure increases the programming complexity, but not the asymptotic running time. ∎

To maintain bead hulls under the operation of splitting the polyline a a vertex, split operations, we must perform computation at several levels: a bead is split into two broken beads or a broken bead is split into a smaller broken bead and a polyline fragment that was contained entirely within the bead; a bead string is split giving a bead string containing the anchor and forcing recomputation of the other, and new tangents are computed for the two bead strings that make up a bead hull. In the next lemma we analyze the cost of splitting a bead hull structure so that all the remaining fragments are contained within beads. We defer the recursive cost of handling fragments within beads until Theorem 4.3.

Lemma 4.2 *Suppose that we are given a polyline with* n *vertices and an on-line sequence of splitting vertices. Then, in* $O(n + (n/k)\log n(\log n + \log^2 k))$ *total time, we can build beads of size* k *and maintain bead hulls under splits for all fragments that are not strictly contained within the original beads.*

Proof: Corollary 3.1 and Lemma 3.3 say that Melkman's algorithm can be used to build the initial beads and initial bead hull in the desired time. We record the history of these computations.

To prove this lemma, we will give bead hulls three types of *credits* with which to pay for all construction and tangent computation after the initialization. We maintain the following invariants: If a bead hull has l (whole) beads to the left of the anchor and r beads to the right, then it has $(l + r)\log(\max\{l, r\})$ *hull credits*. Each unbroken bead has one *bead credit*. And each vertex has up to three *vertex credits*: one if it is inside the convex hull, one if it is inside the convex hull of its string, and one if it is inside the convex hull of its bead. We establish the invariants by giving $(n/k)\log(n/2k)$ hull credits, (n/k) bead credits, and at most $3n$ vertex credits to the initial bead hull.

Each split produces one or two new bead hulls (only one, if one of the fragments is completely contained in an original bead). We assign credits to the new bead hulls according to the invariants. The *credit budget* is the the total number of credits before the split minus the number after the split. We will see that the credit budget is non-negative. We charge $O(\log^2 k)$ computation to each hull credit, $O(\log^2 n)$ to each bead credit, and $O(1)$ to each vertex credit in the budget. Together, these charges establish the lemma.

There are two cases to consider when splitting: either a whole bead is split for the first time or the splitting vertex is contained in one of the broken beads.

Case 1: When a whole bead is split, we charge tangent computation to the bead credit and spend hull credits to rebuild strings of beads. Assume that s whole beads are split off before the anchor; the analysis for splitting after the anchor is symmetric.

The bead hull for the fragment containing the anchor can be obtained in three steps. First, play back the history of Melkman's algorithm to give the string of whole beads between the anchor and splitting vertex. Second, break the bead containing the splitting vertex. Finally, compute tangents from the broken bead to the string, and between the strings before and after the anchor. The first two steps run hull construction algorithms backwards, so we can charge their computation to the initial build. By Lemma 2.2, the third step can be performed in $O(\log^2 n)$ time, which can be charged to the bead credit obtained by breaking a bead. The resulting bead hull must be given

$$(l - s - 1 + r)\log\max\{l - s - 1, r\} \leq (l - s - 1 + r)\log\max\{l, r\}$$

hull credits to satisfy the invariant.

The bead hull for the fragment not containing the anchor must be built from scratch in $O(s\log^2 k)$ time (Lemma 3.3), which consumes s hull credits. Since an anchor is chosen in the middle,

$$s\log\lfloor s/2\rfloor \leq s(\log s - 1) \leq s\log\max\{l, r\} - s$$

hull credits must be given to this fragment.

In case 1, the $(l + r)\log(\max\{l, r\})$ hull credits available are sufficient to pay for the build and satisfy the invariants for the resulting bead hulls. The bead and vertex credits are also sufficient, since splitting decreases the number of unbroken beads by one, and can only increase the total number of hull vertices.

Case 2: When a broken bead is split, we spend vertex credits on updating tangents for the bead hull. No bead or hull credits are spent on computation, as all are needed to maintain the invariants. The new broken bead is formed by playing back the history of the bead's construction, which is charged to the initial construction. Notice that the vertices removed from the broken bead form a fragment that is entirely contained within the original bead. In this lemma, the computation required for such fragments is not considered.

The bead hull has four tangents that may need to be updated. As a representative example, consider the tangent that goes counter-clockwise (ccw) from the string to the broken bead in Figure 6. We shrink the broken bead, playing back the history of Melkman's algorithm, until we reach the splitting vertex. This may cause new vertices to appear on the hull of the broken bead.

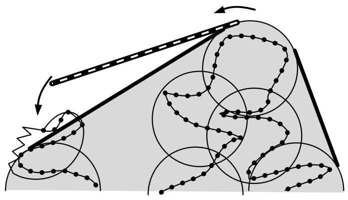

Figure 6: Updating a tangent while shrinking a broken bead

We need to update the tangent if and only if the tangent endpoint is removed. The candidates for the tangent endpoint on the broken bead are the new vertices and those adjacent to them. Candidates on the string are at or ccw of the old tangent endpoint. If we begin by joining the clockwise-most candidates on both bead and string, then by testing incident edges we can determine whether we have found the tangent or which candidate endpoint should move ccw. We can advance until we find a tangent or determine that the broken bead is contained in the hull of the string; we charge the search time to vertex credits taken from vertices that now join the bead and string convex hulls.

When the splitting vertex is common to two beads, then two beads are affected. When the splitting vertex is the anchor, both strings are also affected. These reduce to combinations of cases 1 and 2, however, depending on whether the affected beads were previously whole or broken. ∎

In the previous lemma, it was not necessary that the splits occur at extreme vertices of the bead hull. For applications that do involve finding and splitting at extreme vertices, we can prove the following theorem.

Theorem 4.3 *For a simple polygonal line with n vertices, bead hulls of all fragments can be constructed and maintained under the operations of finding and splitting at extreme vertices in $O(n \log^* n)$ time and $O(n)$ space.*

Proof: By Lemma 4.1, each search for a splitting vertex can be performed in time logarithmic in the size of the smaller fragment. It is known that recursion trees with this behavior take linear time in total [4, 7].

For a polyline of n vertices, let us choose beads of size $k = \log^2 n$. Then Lemma 4.2 says that one level of bead hull computation produces, in time

$$O(n + (n/\log^2 n)\log n(\log n + (\log(\log^2 n))^2)) = O(n),$$

a set of fragments of sizes $n_1, n_2, \ldots, n_f$, with each $n_i < k$ and $\sum_i n_i \le n$.

Let $T(m)$ be an upper bound on the total time to handle a fragment of size m recursively: building beads and maintaining bead hulls until the fragments are of size less than $\log^2 m$. If we handle fragments with less than 512 vertices by a non-recursive method and account for this time separately, then we obtain a base condition of $T(m) = 0$ for $0 < m < 512$; we will show that $T(n) = O(n \log^* n)$ for all $n > 0$.

Since $T(m)$ is a convex function that is at least linear, we can bound $T(n)$ for $n \ge 512$ by a recurrence that is maximized when all fragment sizes, n_i, are as large as possible:

$$T(n) \le O(n) + \sum_{1 \le i \le f} T(n_i) \le O(n) + \frac{n}{\log^2 n} T(\lceil \log^2 n \rceil).$$

We can define a function $t(n)$ by iterating $\log^2$, such that $T(n) = O(nt(n))$:

$$t(n) = \begin{cases} 0 & \text{if } n < 512 \\ 1 + t(\lceil \log n \rceil^2) & \text{otherwise.} \end{cases}$$

Recall that the function $\log^* n$ has a similar definition as the number of times to iterate the logarithm function until its argument is less than 2. The reader can check that $t(n) \le 2\log^* n$ for all $n > 0$ by verifying the inequality for values of n with $t(n) \le 3$, and then observing that

$$\lceil \log^2 \lceil \log^2 n \rceil \rceil \le \lceil \log n \rceil \text{ for all } n > 2^{2^9}.$$

This establishes the total time complexity as $T(n) = O(n \log^* n)$.

Finally, we establish the total memory space required. Any single bead hull data structure takes space proportional to the number of its vertices. If fragments are handled from largest to smallest, then no vertex need participate in more than two fragments at a time. Thus, linear space is sufficient. ■

From the work of Dobkin et al. [1], we obtain an algorithm for building CSG formulæ as a simple corollary.

Corollary 4.4 *CSG formulæ for simple polygons of n vertices can be computed in $O(n \log^* n)$ time and $O(n)$ space.*

The path-hull implementation of the line simplification method in Douglas and Peucker [2] becomes as simple corollary if we add a *side selection rule*, which we will now define. Recall the description of the method from Section 1: to approximate the polygonal line $P = \{p_1, p_2, \ldots, p_n\}$, the two extreme points from the line $\overleftrightarrow{p_1 p_n}$ must be found, and the farther one is used as a splitting vertex. We modify this description as follows: search in parallel on each side of the line $\overleftrightarrow{p_1 p_n}$ until the first extreme point is found. The search information on the other side may tell us that there is a point at greater distance, in which case we continue to find the true extreme point and split there. Or it may say that we already have the extreme point and can split. If, however, the information is inconclusive, then we need a side selection rule that decides, in $O(1)$ time, which side to split on the basis of the information we have so far. Example rules include always splitting on the first extreme point found, always splitting on the second, splitting on the side opposite the last split that formed this fragment, and so forth.

Corollary 4.5 *For simple polylines of n vertices, the Douglas-Peucker line simplification with any side-selection policy can be computed in $O(n \log^* n)$ time and $O(n)$ space.*

The danger of not having a an $O(1)$-time side selection rule is that perhaps the true extreme point is found right away, but more time must be spent to verify that it is the true extreme. This additional cannot be charged against the few vertices that are removed by the split. However, it is difficult, and perhaps impossible, to construct examples in which this happens repeatedly.

5 Conclusion

We have given an $O(n \log^* n)$ algorithm for maintaining a convex hull under splits; this gives a theoretical improvement to the running time for building CSG formulæ for planar polygons and for a modified version of Douglas-Peucker line simplification—showing that both problems are easier than sorting. We expect that the original line simplification procedure is also faster than sorting, but are unable to prove this. The most interesting open problem is whether these problems have practical, linear-time solutions.

6 Acknowledgment

We thank the referees for their careful reading and constructive comments that have improved the presentation of this paper.

References

[1] D. Dobkin, L. Guibas, J. Hershberger, and J. Snoeyink. An efficient algorithm for finding the CSG representation of a simple polygon. *Algorithmica*, 10:1–23, 1993.

[2] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[3] L. Guibas, J. Hershberger, and J. Snoeyink. Compact interval trees: A data structure for convex hulls. *International Journal of Computational Geometry & Applications*, 1(1):1–22, 1991.

[4] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[5] J. Hershberger and J. Snoeyink. Speeding up the Douglas-Peucker line simplification algorithm. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 134–143. IGU Commission on GIS, 1992.

[6] J. Hershberger and J. Snoeyink. An $O(n \log n)$ implementation of the Douglas-Peucker line simplification algorithm. In *Proceedings of the Tenth Annual ACM Symposium on Computational Geometry*, pages 383–384, 1994. Video Review of Computational Geometry. 4:45 animation.

[7] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan sequences in linear time. *Information and Control*, 68:170–184, 1986.

[8] R. B. McMaster. A statistical analysis of mathematical measures for linear simplification. *The American Cartographer*, 13:103–116, 1986.

[9] A. A. Melkman. On-line construction of the convex hull of a simple polyline. *Information Processing Letters*, 25:11–12, 1987.

[10] M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[11] D. P. Peterson. Halfspace representation of extrusions, solids of revolution, and pyramids. SANDIA Report SAND84-0572, Sandia National Laboratories, 1984.

[12] F. P. Preparata and M. I. Shamos. *Computational Geometry—An Introduction.* Springer-Verlag, New York, 1985.

[13] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Vision, Graphics, and Image Processing*, 1:244–256, 1972.

[14] G. Rote. The convergence rate of the Sandwich algorithm for approximating convex functions. *Computing*, 48:337–361, 1992.

[15] E. R. White. Assessment of line-generalization algorithms using characteristic points. *The American Cartographer*, 12(1):17–27, 1985.

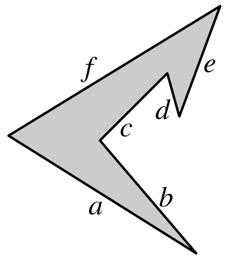

Figure 1: CSG formula is $a(b + ((c + d)e))f$

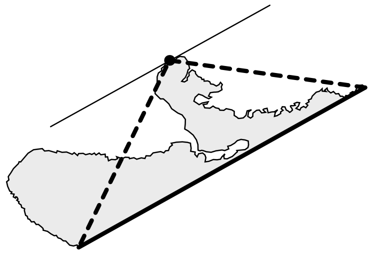

Figure 2: Line simplification

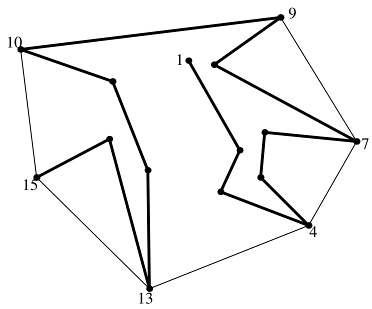

Figure 3: Vertex numbers on the hull

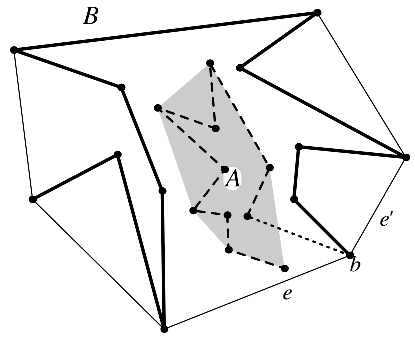

Figure 4: Hull A (shaded) in B

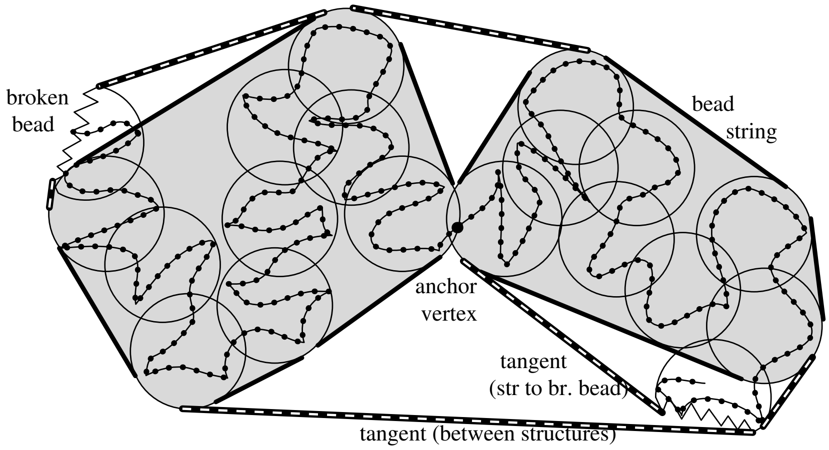

Figure 5: Schematic depiction of a bead hull with anchor in the middle

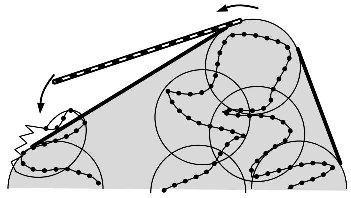

Figure 6: Updating a tangent while shrinking a broken bead

Figure 1: CSG formula is $a(b + ((c + d)e))f$
Figure 2: Line simplification
Figure 3: Vertex numbers on the hull
Figure 4: Hull A (shaded) in B
Figure 5: Schematic depiction of a bead hull with anchor in the middle
Figure 6: Updating a tangent while shrinking a broken bead