Heuristics for the Generation of Random Polygons*

Thomas Auer† Martin Held†

1 Introduction

In this paper we deal with the random generation of simple polygons on a given set of points[1]: Ideally, given a set $\mathcal{S} = \{s_1, \ldots, s_n\}$ of n points, we would like to generate a simple polygon $\mathcal{P}$ at random with a uniform distribution. In this context, a uniformly random polygon on $\mathcal{S}$ is a polygon which is generated with probability $\frac{1}{k}$ if there exist k simple polygons on $\mathcal{S}$ in total. Since no polynomial-time solution is known for the uniformly random generation of simple polygons, we focus on heuristics that offer a good time complexity and still generate a rich variety of different polygons.

Besides being a topic of interest of its own, the generation of random polygons has two main areas of application: a) testing the correctness and b) evaluating the CPU-time consumption of algorithms that operate on polygons. Ideally, one would like to apply an algorithm on data of practical relevance. However, it often is next to impossible to obtain enough practically relevant inputs. Then the second-best choice is to run an algorithm for a reasonably large number of random inputs.

Recently, the generation of random geometric objects has received some attention by researchers. For example, Epstein [Eps92] studied the uniformly random generation of triangulations. Zhu et al. [ZSSM96] presented an algorithm for generating x-monotone polygons on a given set of vertices uniformly at random. A heuristic for the generation of simple polygons was investigated by O'Rourke and Virmani [OV91]. Note, however, that their algorithm moves the vertices while creating a polygon.

In Section 2, we study the following five heuristics for the random generation of simple polygons:

Steady Growth, an incremental algorithm adding one point after the other;

Space Partitioning, which is a divide and conquer algorithm;

Permute & Reject, which creates random permutations (polygons) until a simple polygon is encountered;

2-opt Moves, which generates a random (non-simple) polygon and repairs the deficiencies; and

Incremental Construction & Backtracking, which tries to minimize backtracking by eliminating dead search trees.

All these algorithms have been implemented and subjected to extensive testing In Section 3, we report and analyze the results obtained.

In addition, we study Star Universe which is an algorithm for the enumeration of all star-shaped polygons on a given point set. Star Universe is compared to a fast heuristic, Quick Star.

Throughout this paper we assume that the vertices $v_1, \ldots, v_n$ of an n-gon $\mathcal{P}$ are specified in counterclockwise (CCW) order. A point of the input set[2] $\mathcal{S}$ is denoted by s, whereas p stands for an arbitrary point in the plane. In our algorithms, $\mathcal{P}_i$ denotes the polygon obtained after the execution of phases 1 through i of the algorithm. For a set $\mathcal{Q}$, we denote its convex hull by $\mathcal{CH}(\mathcal{Q})$. We let $\ell(p_1, p_2)$ stand for a line through p_1,p_2, and the restriction to the line segment is denoted by $\overline{p_1 p_2}$.

2 Algorithms

2.1 Star-Shaped Polygons

Star Universe We start with explaining how to enumerate all star-shaped polygons on $\mathcal{S}$. Obviously, a star-shaped polygon $\mathcal{P}$ is fixed once its kernel has been specified: For every point p that lies within the kernel, the polygon's vertices appear in sorted order around p. Also, this order is identical for every point interior to the kernel. Therefore, the kernels

*A draft of the full paper is available on the WWW at http://www.cosy.sbg.ac.at/~held/publications.html.

†Universität Salzburg, Computerwissenschaften, A-5020 Salzburg, Austria. E-mail: {tom,held}@cosy.sbg.ac.at.

[1] Since no generally accepted definition for random polygons in the plane exists, we restrict ourselves to the study of random polygons on a given set of points.

[2] For the sake of descriptional simplicity, we assume that $\mathcal{S}$ is in general position.

of two distinct star-shaped polygons share at most one edge, and the set of all kernels forms a valid partition of $\mathcal{CH}(\mathcal{S})$.

Necessarily, the arrangement induced by all lines $\ell(s_i, s_j)$, where $1 \leq i < j \leq n$, contains all the kernels. In general, several adjacent cells of the arrangement will belong to one kernel. This arrangement contains at most $\mathcal{O}(n^4)$ cells and it can be computed in $\mathcal{O}(n^4)$. All the kernels can be constructed from the arrangement in time linear in its size by the following depth-first search:

1. Choose one cell and mark it as visited.

2. Create the star-shaped polygon $\mathcal{P}$ defined by this "active" cell.

3. For each edge e on the boundary of the active cell, do the following: If the neighboring cell bounded by e is part of the same kernel, then check its edges using $\mathcal{P}$. Otherwise, (i.e., if the neighboring cell belongs to another kernel) invert the order of the vertices that lie on the supporting line of e, and proceed with this other cell (and the modified polygon).

Note that the neighboring cell belongs to the same kernel if and only if the two vertices defining e do not appear in consecutive order in $\mathcal{P}$.

Since the method outlined above consumes $\mathcal{O}(n^4)$ space[3] independent of the actual number of star-shaped polygons, we also investigated the following output-sensitive method dubbed Star Universe: For each line $\ell(s_i, s_j)$, we compute the intersections with all other lines (defined by pairs of points of $\mathcal{S}$) and sort them according to their intersection parameters[4]. (We can restrict $\ell(s_i, s_j)$ to the portion that lies within $\mathcal{CH}(\mathcal{S})$.) For each intersection point p we keep track of the line ℓ which generated it. For the first[5] star-shaped polygon $\mathcal{P}$, we compute the midpoint of the first two intersections and sort the points of $\mathcal{S}$ around this midpoint. Then we process each intersection point p, starting with the second one, as follows: If the line ℓ associated with p coincides with an edge of $\mathcal{P}$ then we swap the points defining ℓ, thus updating $\mathcal{P}$. Otherwise, we skip p.

Note that polygons may be encountered more than once during the execution of this algorithm. Thus, we have to keep track of all the polygons generated so far, which can be done in more than one way (depending on the desired trade-off between time and space complexity).

[3]If all k resulting polygons are to be stored, the space complexity goes up to $\mathcal{O}(n^4 + n \cdot k)$.

[4]I.e., sort them according to their x-coordinate (or according to their y-coordinate if the line is vertical).

[5]Note that if one of the edges of $\mathcal{P}$ coincides with ℓ, we actually get two polygons.

Star Universe can be implemented with a time complexity of $\mathcal{O}(n^5 \log n)$ as opposed to $\mathcal{O}(n^4)$ for the previous algorithm. However, the space requirement is reduced from $\mathcal{O}(n^4 + n \cdot k)$ to $\mathcal{O}(n^2 + n \cdot k)$ space[6], where k denotes the number of star-shaped polygons to be stored. (See Section 3 for experimental results on k.)

Quick Star Every point p which lies within $\mathcal{CH}(\mathcal{S})$ defines a star-shaped polygon. Therefore, the following simple method dubbed Quick Star generates every possible star-shaped polygon with positive probability: Choose a random point p within $\mathcal{CH}(\mathcal{S})$, and sort the points of $\mathcal{S}$ around p.

We use a rejection method in order to generate a point within $\mathcal{CH}(\mathcal{S})$ with uniform distribution: Choose a point within the bounding box of $\mathcal{CH}(\mathcal{S})$, until one which actually lies within $\mathcal{CH}(\mathcal{S})$ is found.

2.2 Simple Polygons

Steady Growth As initialization, Steady Growth randomly selects three points $s_1, s_2, s_3 \in \mathcal{S}$ such that no other point of $\mathcal{S}$ lies within $\mathcal{CH}(\{s_1, s_2, s_3\})$. Let $\mathcal{S}_1 := \mathcal{S} \setminus \{s_1, s_2, s_3\}$. During the i-th iteration (with $1 \leq i \leq n-3$), we

1. Choose one point $s_i \in \mathcal{S}_i$ at random such that no remaining point of $\mathcal{S}_{i+1} := \mathcal{S}_i \setminus \{s_i\}$ lies within $\mathcal{CH}(\mathcal{P}_{i-1} \cup \{s_i\})$.

2. Find an edge (v_k, v_{k+1}) of $\mathcal{P}_{i-1}$ that is completely visible from s_i, and replace it with the edges (v_k, s_i) and (s_i, v_{k+1}).

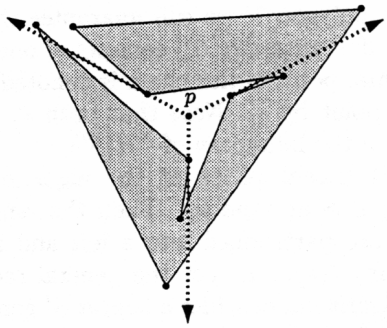

Figure 1: The point p does not see any edge of $\mathcal{P}$ completely.

[6]When applied to 50 points the process size of the arrangement-based code grew up to 236MB whereas Star Universe consumed only 16MB of main memory. Due to swapping, the arrangement-based code was significantly slower than Star Universe, too.

Note that a point $s_i \in \mathcal{S}_i$ which is suitable for Step 1 always exists. It is less straightforward to guarantee that a suitable edge always exists in Step 2. As illustrated in Fig. 1, a point p which lies outside a general polygon $\mathcal{P}$ need not see any edge of $\mathcal{P}$ completely. However, if p lies outside $\mathcal{CH}(\mathcal{P})$, there must exist at least one edge which is completely visible from p. This can be shown by induction: Compute the supporting vertices of $\mathcal{CH}(\mathcal{P})$, and consider the chain from the left supporting vertex to the right one. (This is the chain which faces p.) If this chain consists of only one edge (defined by the two supporting vertices), then it must be completely visible since both its endpoints are visible. Otherwise, if the chain consists of k edges, then consider its leftmost[7] edge e_l. We are done if this edge is completely visible. Otherwise, consider the leftmost edge e'_l which is in front of e_l (and which faces p). Necessarily, the left endpoint of e'_l must be visible from p. Thus, we obtain a new chain with at most $k-1$ edges whose left and right endpoints are visible from p.

By using Steady Growth, one can compute a simple polygon in at most $\mathcal{O}(n^2)$ time, since all that has to be done during each phase is to compute all edges which are completely visible. (This can be done in $\mathcal{O}(n)$ time, cf. Joe and Simpson [JS87].) Note that selecting a suitable point s_i in Step 1 can be carried out in linear time, too. Unfortunately, Steady Growth does not generate every possible polygon.

Space Partitioning Space Partitioning recursively partitions $\mathcal{S}$ into subsets which have disjoint convex hulls. Let $\mathcal{S}'$ be a such a subset of $\mathcal{S}$. (Thus, $\mathcal{CH}(\mathcal{S}')$ does not contain any point of $\mathcal{S} \setminus \mathcal{S}'$.) When generating a polygon $\mathcal{P}$ we will guarantee that the intersection of $\mathcal{P}$ with $\mathcal{CH}(\mathcal{S}')$ consists of one single chain. The first point of this chain is denoted by s'_f, and its last point by s'_l. Note that both s'_f and s'_l are located on the boundary of $\mathcal{CH}(\mathcal{S}')$.

During the initial phase of the algorithm, we choose $s_f, s_l \in \mathcal{S}$ at random. Then the remaining points of $\mathcal{S}$ are partitioned into a left and a right set by the line $\ell(s_f, s_l)$. For the general recursive call of the algorithm, consider a subset $\mathcal{S}'$ generated by this recursive subdivision, and let s'_f be its first point and s'_l its last point. If s'_f and s'_l are the only points of $\mathcal{S}'$, then the line segment $\overline{s'_f s'_l}$ is output and the recursion is terminated. Otherwise, in order to split $\mathcal{S}'$ into two subsets $\mathcal{S}''$ and $\mathcal{S}'''$, we

1. Pick a point $s' \in \mathcal{S}'$ at random.

[7] I.e., the edge whose endpoint has the smallest angular distance from the left supporting vertex.

2. Select a random line ℓ through s' such that ℓ intersects $\overline{s'_f s'_l}$. The line ℓ splits $\mathcal{S}'$ into two subsets $\mathcal{S}''$ and $\mathcal{S}'''$, where $\mathcal{S}''$ has s'_f as its first and s' as its last point, cf. Fig. 2. Similarly, $\mathcal{S}'''$ has s' as its first and s'_l as its last point.

Since $\mathcal{S}''$ and $\mathcal{S}'''$ lie on opposite sides of ℓ, $\mathcal{CH}(\mathcal{S}'')$ and $\mathcal{CH}(\mathcal{S}''')$ are disjoint. Furthermore, $\mathcal{CH}(\mathcal{S}'')$ and $\mathcal{CH}(\mathcal{S}''')$ do not contain any point of $\mathcal{S} \setminus \mathcal{S}'$.

Unfortunately, Space Partitioning does not generate every possible polygon on $\mathcal{S}$.

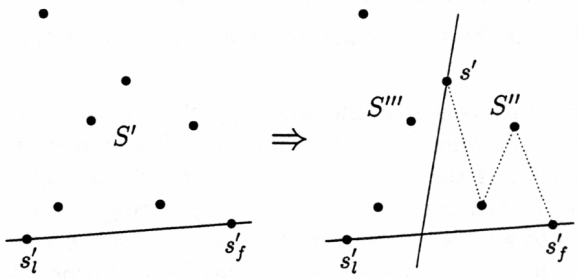

Figure 2: Partition of set $\mathcal{S}'$ and a sample path through $\mathcal{S}''$.

Permute & Reject For Permute & Reject, we create a permutation of $\mathcal{S}$ and check whether this permutation corresponds to a simple polygon. If the polygon is simple then it is output; otherwise a new polygon is generated.

Obviously, the actual running time of this method mainly depends on how many polygons need to be generated in order to encounter a simple polygon. (In the next section, we report experimental results.) Clearly, Permute & Reject produces all possible polygons with a uniform distribution.

2-opt Moves This approach first generates a random permutation of $\mathcal{S}$, which again is regarded as the initial polygon $\mathcal{P}$. Any self-intersections of $\mathcal{P}$ are removed by applying so-called 2-opt moves. Every 2-opt move replaces a pair of intersecting edges (v_i, v_{i+1}), (v_j, v_{j+1}) with the edges (v_{j+1}, v_{i+1}) and (v_j, v_i). In our application, at each iteration of the algorithm one pair of intersecting edges is chosen at random and the intersection is removed.

Van Leeuwen and Schoone [vLS82] showed that at most $\mathcal{O}(n^3)$ many 2-opt moves need to be applied in order to obtain a simple polygon. Thus, an overall time complexity of $\mathcal{O}(n^4)$ can be achieved. As explained in Zhu et al. [ZSSM96], 2-opt Moves will

produce all possible polygons, but not with a uniform distribution.

Incremental Construction & Backtracking Finally we studied an approach based on exhaustive search and backtracking which is akin to the work by Shuffelt and Berliner [SB94]. We start with a polygonal chain consisting of one randomly chosen point, and we randomly add one point after the other to it as long as the resulting chain remains simple. Backtracking has to be applied when a non-simple chain is encountered. Clearly, the main crux is to avoid any extensive backtracking.

In order to reduce backtracking, we keep an inventory of those edges which still are usable for completing the polygon. (Edges which are no longer usable get marked.) Initially, all edges of the complete graph on $\mathcal{S}$ are usable. When adding point s, and thus using some edge e, all the edges that intersect e are marked. Furthermore, if a point is adjacent to two other points that both have only two incident unmarked edges, we mark all the other edges incident upon that point. Clearly, backtracking is necessary if any of the following conditions is violated:

1. Each point that does not yet belong to the polygonal chain under construction has at least two incident unmarked edges.

2. At most one point adjacent to the point last added has only two incident unmarked edges.

3. Points that lie on the boundary of $\mathcal{CH}(\mathcal{S})$ appear in the polygonal chain in the same relative order as on the hull.

This algorithm produces every possible simple polygon with positive probability. Clearly, its efficiency depends on the amount of backtracking necessary. (See next section.)

3 Practical Aspects

3.1 Implementation

We implemented our algorithms together with a test bed in the programming language C. For the generation of random numbers we used *rand48* of the standard C library.

In order to keep the implementation simple and still be able to handle data without the underlying assumption of general position, our implementation differs slightly from the description given above.

Star-Shaped Polygons For Star Universe, we resort the vertices of the polygon (expressed in polar coordinates with respect to p) each time we enter a new kernel, thus simplifying the cumbersome handling of multiple collinear points. In Quick Star, we always sort points that have the same polar angle with respect to the random point p (which belongs to the polygon's kernel) by their distance from p. Thus, our implementation may miss some star-shaped polygons on sets with multiple collinear points.

Simple Polygons For calculating the edge visibilities in Steady Growth, we did not implement the linear algorithm by Joe and Simpson [JS87], but simply sorted the vertices around the point to add.

3.2 Experimental Results

We ran three different series of experiments, which are reported in the following paragraphs: First, we recorded the CPU-consumption of our algorithms. Second, we obtained experimental bounds on the numbers of star-shaped and simple polygons in terms of the cardinality of the point set. Third, we evaluated the number of polygons generated by our algorithms in order to assess the quality and practical applicability of these heuristics. (All tests were run on Sun SPARCstations 20.)

CPU-Time Consumption We measured the CPU-time consumption of each algorithm when applied to random point sets (within the unit square) of the following cardinalities: 10, 25, 50, 100, 200, 300, 400 and 500. For each of these cardinalities we generated three independent sets. Our algorithms had to compute 50 polygons on each of these sets. The mean elapsed CPU-times (in milliseconds) were plotted on a logarithmic scale ($\log_2 t$).

As expected, Star Universe is only feasible for input sets with a small cardinality, cf. Fig. 3: Our attempts to run Star Universe on 100 points had to be aborted due to lack of main memory. (192MB did not suffice.) Quick Star, however, seems well suited for larger point sets: computing a star-shaped polygon on 500 points takes roughly 62 milliseconds.

Two of the algorithms for the generation of simple polygons are not applicable to anything but extremely small point sets, cf. Fig. 3: In more than three weeks of running time we were not able to generate results for 25 points when using Permute & Reject or Incremental Construction & Backtracking.

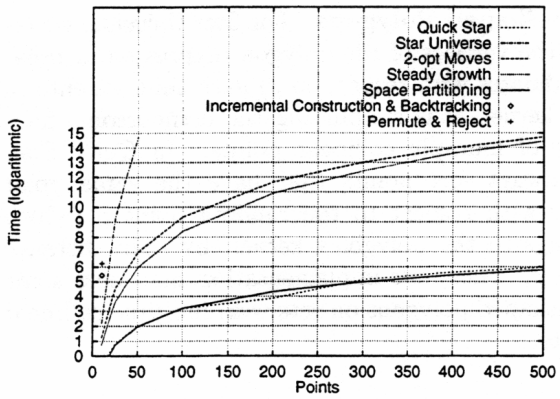

Figure 3: CPU-consumption of the algorithms.

Among the remaining three methods, Space Partitioning is significantly faster than the two other algorithms. Roughly, Space Partitioning takes about 55 milliseconds to compute a simple polygon on 500 points, whereas Steady Growth and 2-opt Moves consume about 25 seconds. Note that Quick Star and Space Partitioning consume about the same amount of CPU time.

Number of Polygons By using a modified version of Incremental Construction & Backtracking, we determined the number of simple polygons on groups of ten sets with 10 respectively 15 random points. (Due to lack of space, no test results for sets of 15 points are included in this abstract; see the full paper for details.) For star-shaped polygons, we enumerated all polygons for groups of ten sets with 10, 15, 20, 25 and 50 points each by means of Star Universe. All these numbers are listed in Table 1. In our tests, all polygons which describe the same geometric figure were counted exactly once. Note that test runs for counting all simple polygons on 20 points crashed due to lack of disk space after generating more than three million different polygons.

$\lvert S_i \rvert$	Simple		Star-Shaped				
	10	15	10	15	20	25	50
1	351	195,554	67	320	1,061	2,666	59,017
2	329	58,768	51	266	1,015	2,340	77,685
3	164	65,338	44	287	995	3,318	63,741
4	776	291,232	103	516	1,816	4,120	82,478
5	146	149,701	44	358	1,170	2,827	66,708
6	321	269,022	57	435	1,450	3,696	71,943
7	852	199,266	76	418	1,447	3,906	70,147
8	346	150,423	50	357	1,136	3,203	67,213
9	380	281,324	56	382	1,293	3,680	64,466
10	599	205,536	87	392	1,353	2,916	65,004

Table 1: No. of simple and star-shaped polygons.

Quality Assessment For each algorithm, we determined the ratio of the number of polygons generated and the total number of possible polygons experimentally. When generating 100,000 star-shaped polygons on 20 points with Quick Star, the mean percentage of polygons hit at least once was 91.439 with a minimum of 89.250 and a maximum of 94.679. When generating 10,000 polygons on 20 points we got 65.361 as mean, 59.141 as minimum and 69.241 as maximum. For 25 points and 100,000 polygons generated, we got a mean of 84.045, a minimum of 80.461 and a maximum of 87.634, whereas the corresponding numbers for 10,000 polygons are 51.769, 44.830, and 55.085. Since Quick Star is capable of producing all possible star-shaped polygons it does not come as a big surprise that the hit rate goes up as the number of polygons generated is increased.

Among the methods for simple polygons, one method is significantly worse than the others: Incremental Construction & Backtracking. As can be seen in Fig. 4, less than half of all possible simple polygons are hit at least once when generating 10,000 polygons on 10 points. As could be expected, Permute&Reject exhibits an optimal hit rate of 100%. For the three algorithms with a modest CPU-consumption, the results are good for 2-opt Moves, acceptable for Steady Growth, but rather poor for Space Partitioning. Note that the results improved when generating 100,000 polygons instead of 10,000 polygons, cf. Fig. 5: 2-opt Moves generates almost all polygons, Steady Growth lies around or above 90%, and Space Partitioning generates about 80–90% of all possible polygons. In both tests the distribution of the polygons turned out to be highly non-uniform, though.

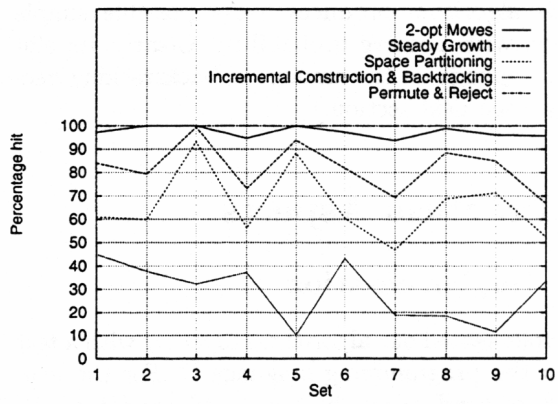

Figure 4: 10,000 simple polygons on 10 points.

However, when generating 100,000 polygons on 100 points all three algorithms 2-opt Moves, Steady

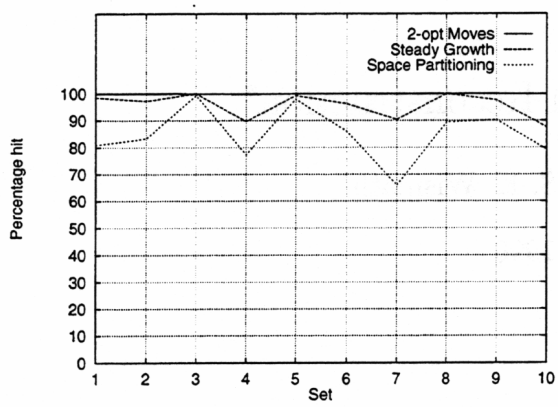

Figure 5: 100,000 simple polygons on 10 points.

Growth and Space Partitioning behaved optimally: They all generated exactly 100,000 different polygons! It is likely that this result is due to the fact that there exists an enormous number[8] of simple polygons on 100 points.

On one hand, our tests[9] suggest that a user of any of these three algorithms need not worry about repeatedly generating the same "random" polygons when dealing with 100 or more points. On the other hand, the distribution of the polygons generated should not be expected to be (close to) uniform.

4 Conclusion

Summary We presented five heuristics for the random generation of simple polygons. Three of them, namely 2-opt Moves, Steady Growth and Space Partitioning, are suited for practical purposes. However, for sets with 10–15 points we experienced a clear trade-off between speed and quality. When the CPU-consumption is not at a premium, one can afford to generate a fairly large variety of polygons by 2-opt Moves. In order to achieve maximum speed Space Partitioning would be the method of choice. Steady Growth is slightly faster than 2-opt Moves but generates a less rich set of polygons. Quick Star (for the generation of star-shaped polygons) has about the same characteristics as Space Partitioning.

For sets of 100 or more points, the class of simple polygons is rich enough and the power of these three heuristics is large enough that any of them can be expected to yield fairly good results. In particular, it is fairly unlikely that a simple polygon will be generated repeatedly by any of these three heuristics. The same comments apply to Star Universe for generating star-shaped polygons.

Open Problems In order to enhance our statistical analysis, we would need to circumvent time and space constraints imposed by the hardware on the enumeration of all simple (respectively, of all star-shaped) polygons on $\mathcal{S}$. Also, it would be desirable to classify in an intuitive manner the classes of polygons that are likely to be generated by our heuristics.

From a theoretical point of view, it remains an open problem to generate polygons on a given set of vertices uniformly at random. From a practical point of view, it is not even clear what constitutes a good "random" polygon. A typical user may want to generate "random" polygons within some fuzzy subclass of polygons: e.g., generate random polygons which consist of two dominant "roughly convex" regions linked by a "roughly x-monotone" tunnel.

Acknowledgments

We thank Joseph Mitchell and Karel Zikan for interesting discussions on this paper's topic.

References

[Eps92] P. Epstein. Generating Geometric Objects at Random. Master's thesis, CS Dept., Carleton University, Ottawa K1S 5B6, Canada, 1992.

[JS87] B. Joe and R.B. Simpson. Corrections to Lee's Visibility Polygon Algorithm. *BIT*, 27:458–472, 1987.

[OV91] J. O'Rourke and M. Virmani. Generating Random Polygons. Technical Report 011, CS Dept., Smith College, Northampton, MA 01063, July 1991.

[SB94] J.A. Shuffelt and H.J. Berliner. Generating Hamiltonian Circuits Without Backtracking from Errors. *Theoretical Comput. Sci.*, 132:347–375, 1994.

[vLS82] J. van Leeuwen and A.A. Schoone. Untangling a Travelling Salesman Tour in the Plane. In J.R. Mühlbacher, editor, *Proc. 7th Conf. Graph-theoretic Concepts in Comput. Sci. (WG 81)*, pages 87–98, 1982.

[ZSSM96] C. Zhu, G. Sundaram, J. Snoeyink, and J.S.B. Mitchell. Generating Random Polygons with Given Vertices. *Comput. Geom. Theory and Appl.*, to appear 1996.

[8] We encountered sets with 20 points which already allowed more than three million simple polygons.

[9] Any further statistical analysis of the distributions of the polygons generated had to be abandoned due to hardware constraints imposed on the CPU-time consumption and the available main memory and disk space.