RPG – Heuristics for the Generation of Random Polygons

Thomas Auer*
Institut für Computerwissenschaften
Universität Salzburg
A–5020 Salzburg, Austria

Martin Held†
Dept. of Applied Mathematics
SUNY Stony Brook
Stony Brook, NY 11794-3600, U.S.A.

Abstract

We consider the problem of randomly generating simple and star-shaped polygons on a given set of points. This problem is of considerable importance in the practical evaluation of algorithms that operate on polygons, where it is necessary to check the correctness and to determine the actual CPU-consumption of an algorithm experimentally.

Since no polynomial-time solution for the uniformly random generation of polygons is known, we present and analyze several heuristics. All heuristics described in this paper have been implemented and are part of our RandomPolygonGenerator, RPG. We have tested all heuristics, and report experimental results on their CPU-consumption, their quality, and their characteristics. RPG is publically available via http://www.cosy.sbg.ac.at/~held/projects/rpg/rpg.html.

1 Introduction

In this paper[1] we deal with the random generation of simple polygons on a given set of points: 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 such that the points of $\mathcal{S}$ form the vertices of $\mathcal{P}$. 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. For testing the correctness of an algorithm, the goal is to generate a diverse set of input data such that all branches of the algorithm will be executed with a high probability. The same motivation applies to the practical testing of an algorithm's CPU-consumption. Ideally, one would like to test the performance of an algorithm on data of practical relevance. However, it often is next to impossible to obtain a sufficiently large number of 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. Thus, it does not fall into the class of algorithms presented in this paper.

In the sequel, we analyze the following five heuristics for the 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;

*Email: tom@cosy.sbg.ac.at.
†Email: held@ams.sunysb.edu. Part of this work was supported by grants from Boeing Computer Services, and by NSF Grants DMS-9312098 and CCR-9504192. The author is on leave from Institut für Computerwissenschaften, Universität Salzburg, A–5020 Salzburg, Austria.

[1] A preliminary version of this work appeared in [AH96].

- 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, and 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. This enumeration can be used for generating a random star-shaped polygon with a uniform distribution. Star Universe is compared to a fast heuristic, Quick Star.

All algorithms described in this paper are part of our software package RPG, which is the acronym for RandomPolygonGenerator. RPG offers an easy-to-use graphical user interface, which is based on the Forms Library by Zhao and Overmars [ZO95]. RPG can be retrieved via anonymous ftp from its WWW home-page at URL: http://www.cosy.sbg.ac.at/~held/projects/rpg/rpg.html. (In addition to the source code distribution, executables for a few UNIX platforms are provided, too. See the README-file for more details.)

Throughout this paper we assume that the n vertices $v_1, \ldots, v_n$ of a polygon ("n-gon") $\mathcal{P}$ are specified in counterclockwise (CCW) order. A point of the input set $\mathcal{S}$ is denoted by s, whereas p stands for an arbitrary point in the plane. For the sake of descriptional simplicity, we assume that $\mathcal{S}$ is in "general position". ("General position" means that no three points of $\mathcal{S}$ are collinear, and that no four points are cocircular. Note that our implementation does support multiple collinear or cocircular points.)

In our algorithms, $\mathcal{P}_i$ stands for the polygon obtained after the execution of phases 1 through i of the algorithm. The convex hull of a set $\mathcal{Q}$ is denoted 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 $\overline{p_1 p_2}$.

This paper is organized as follows: In Section 2, we describe the basics of our algorithms. Section 3 deals with implementational issues and discusses the experimental results obtained. We conclude our paper in Section 4.

2 Algorithms

2.1 Generating Star-Shaped Polygons

2.1.1 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 order of the polygon's vertices is fixed because the vertices appear in sorted order around p. Also, this order is identical for every point interior to the kernel. Therefore, the kernels of two distinct star-shaped polygons share at most one edge, and the set of all kernels forms a valid partition of the convex hull[2] $\mathcal{CH}(\mathcal{S})$.

Thus, for enumerating all star-shaped polygons, it is sufficient to compute this partition of the convex hull. 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: In Fig. 1, the kernel of polygon a) occupies the cells 1–8, the kernel of polygon b) occupies the cells 9–11, cell 12 corresponds to the kernel of polygon c), and the cells 13–15 give the kernel of polygon d).

The number of all lines $\ell(s_i, s_j)$ is bound by $\mathcal{O}(n^2)$, and therefore the arrangement contains at most $\mathcal{O}(n^4)$ cells, i.e., at most $\mathcal{O}(n^4)$ kernels. Thus, there exist at most $\mathcal{O}(n^4)$ star-shaped polygons[3] on a set of n points. The arrangement can be computed in time $\mathcal{O}(n^4)$, cf. [O'R94]. All 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 $\mathcal{P}$.

[2] Trivially, all kernels are confined to the convex hull.

[3] This bound was derived independently by [BS96]. It is not known whether this bound is tight.

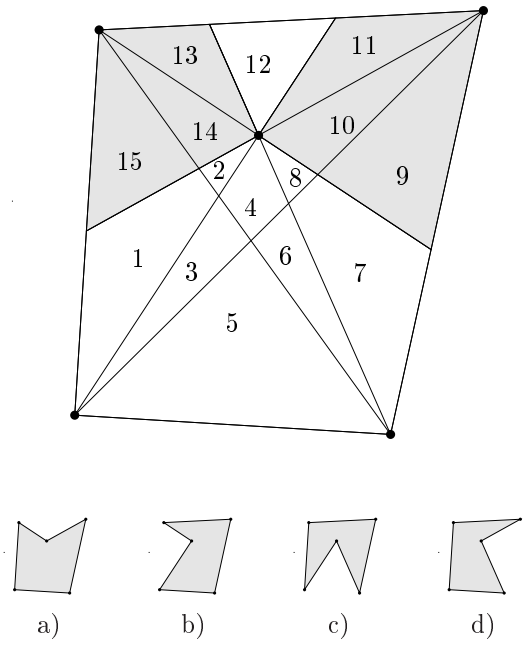

Figure 1: Star-shaped polygons and their kernels.

Since the method outlined above consumes $\mathcal{O}(n^4)$ space[4] 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[5]. Since all kernels lie within $\mathcal{CH}(\mathcal{S})$, 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[6] 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). We opted for actually storing all polygons in an AVL-tree.

Algorithm 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[7], where k again denotes the number of star-shaped polygons to be stored. (See Section 3 for experimental results on k.)

Naturally, the algorithms outlined above can also be used for generating a star-shaped polygon uniformly at random: First, enumerate all k existing star-shaped polygons on $\mathcal{S}$, then choose one at random.

2.1.2 Quick Star

Every point p which lies within $\mathcal{CH}(\mathcal{S})$ defines a star-shaped polygon. (If p lies on an edge or coincides with a vertex of the arrangement, up to four[8] star-shaped polygons are defined.) 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. Clearly, this approach requires $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.

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

2.2 Generating Simple Polygons

2.2.1 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 $\mathcal{CH}(\mathcal{P}_{i-1} \cup \{s_i\})$.

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

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

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

[7] When applied to 50 points the process size of the arrangement-based code grew up to 236MB (and the program execution ended up in frequent swapping) whereas Star Universe consumed only the modest amount of 16MB of main memory. Due to swapping, the arrangement-based code was significantly slower than Star Universe, too. Unfortunately, as explained in Section 3, the memory demand of Star Universe still is too high for practical purposes.

[8] Recall that we assume $\mathcal{S}$ to be in general position.

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}).

Note that a point $s_i \in \mathcal{S}_i$ which is suitable for Step 1 always exists. (For example take the point that lies closest to $\mathcal{CH}(\mathcal{P}_{i-1})$.)

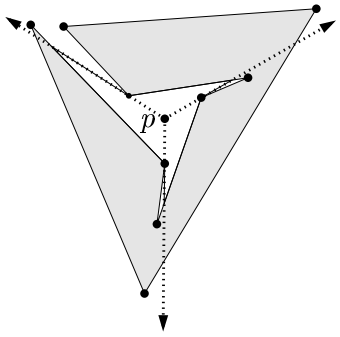

Figure 2: The point p does not see any edge of $\mathcal{P}$ completely.

It is less straightforward to guarantee that a suitable edge always exists in Step 2. As illustrated in Fig. 2, a point p which lies outside a general polygon $\mathcal{P}$ need not see any edge of $\mathcal{P}$ completely. (A similar counterexample exists for p within $\mathcal{P}$.) However, if p lies outside $\mathcal{CH}(\mathcal{P})$ then there exists at least one edge which is completely visible from p. This claim 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[9] 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 on $\mathcal{S}$, cf. [Aue96].

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

2.2.2 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. (Recall that $\mathcal{CH}(\mathcal{S}')$ does not contain any point of $\mathcal{S} \setminus \mathcal{S}'$.) 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.
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. 3. 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 $\mathcal{S} \setminus \mathcal{S}'$.

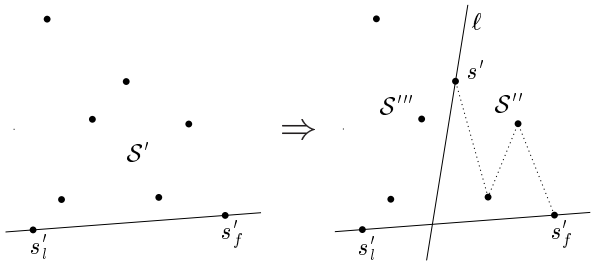

Figure 3: Partition of set $\mathcal{S}'$ and a sample path through $\mathcal{S}''$.

In the worst case, this algorithm executes $\mathcal{O}(n)$ recursive calls, which results in $\mathcal{O}(n^2)$ time. However, if the recursive subdivision is somewhat balanced then we get a time behavior of roughly $\mathcal{O}(n \log n)$. Unfortunately, Space Partitioning does not generate every possible polygon on $\mathcal{S}$, cf. [Aue96].

2.2.3 Permute & Reject

For Permute & Reject, we create a permutation[10] 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. In theory, the test for simplicity can be carried out in linear time by using Chazelle's [Cha91] triangulation algorithm. A more practical approach would be to use an $\mathcal{O}(n \log n)$ algorithm, cf. [PS90].

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. (We report experimental results in the next section.) Clearly, Permute & Reject produces all possible polygons with a uniform distribution.

2.2.4 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), cf. Fig. 4. 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.

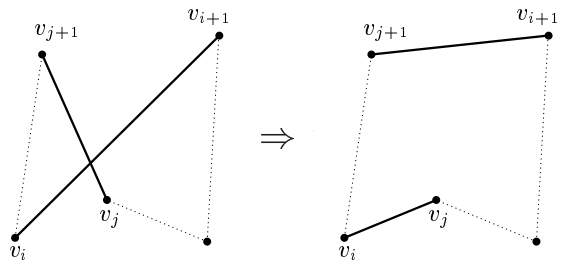

Figure 4: An example for a 2-opt move.

2-opt Moves will produce all possible polygons, but not with a uniform distribution: As explained in Zhu et al. [ZSSM96], there exist polygons which are obtained from the initial polygon by a unique series of 2-opt moves, whereas other polygons may be obtained from the initial polygon by several different series of 2-opt moves.

2.2.5 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 because they are no longer usable for completing the polygon. Furthermore, if a point is adjacent[11] 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. (Otherwise, it is impossible to add this point and still complete the polygon.)

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.

These conditions are checked in the same order as stated.

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

[10] I.e., we arrange the indices of the n input points as an n-tuple, and compute a random permutation of the indices; a random permutation can be obtained in $\mathcal{O}(n)$, cf. Knuth [Knu69].

[11] Two points are called "adjacent" if they are linked by an unmarked edge.

3 Practical Aspects and Experiments

3.1 Implementation

We implemented our algorithms[12] together with a test bed in the programming language C. For the generation of random numbers we used *rand48*[13], which belongs to the standard C library. Our test bed was enhanced by a graphical user interface based on the Forms Library by Zhao and Overmars [ZO95].

RPG handles input and output in floating-point format. Also, all numerical calculations are based on standard floating-point arithmetic. We emphasize that we have not experienced any robustness problems in our use of RPG. However, note that the simplicity of the polygons generated by RPG is only guaranteed up to the limits imposed by floating-point arithmetic on the determination of the sign of determinants.

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 in the previous section. The major differences are highlighted in the following two paragraphs.

3.1.1 Star-Shaped Polygons

Our algorithms for generating star-shaped polygons were implemented with the following changes:

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.

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.

3.1.2 Simple Polygons

Our algorithms for generating simple polygons were implemented with the following changes:

Steady Growth: For calculating the edge visibilities, we did not implement the linear algorithm by Joe and Simpson [JS87], but simply sorted the vertices around the point to be added, thus using $\mathcal{O}(n \log n)$ time.

Permute & Reject: The simplicity test for a polygon is not done in linear time; rather, we implemented a straightforward quadratic approach. (As we will see later this had no influence on the test results.)

3.2 Experimental Results

We ran four different series of experiments, which are reported in the following paragraphs.

1. We recorded the CPU-time consumption of our algorithms.
2. We obtained experimental bounds on the numbers of star-shaped and simple polygons in terms of the cardinality of the point set.
3. We evaluated the number of polygons generated by our algorithms in order to assess the quality and practical applicability of these heuristics.
4. We obtained statistics on some basic characteristics of the polygons generated: we recorded their sinuosities and examined the distributions of their internal angles.

All tests were carried out on Sun SPARCstations 20 running SunOS 5.5. For our tests, we used diverse sets of points formed by generating random points within the unit square[14].

3.2.1 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 using a logarithmic scale ($\log_2 t$).

As expected, Star Universe is only feasible for input sets with a small cardinality, cf. Fig. 5: Our attempts to run Star Universe on 100 points had to be aborted due to lack of main memory[15]. Quick Star, however, seems well suited for larger point sets: computing a star-shaped polygon on 500 points takes roughly 62 milliseconds.

[12] We also implemented the algorithms of Zhu et al. [ZSSM96] and O'Rourke and Virmani [OV91].

[13] *rand48* is based on a linear congruential algorithm and on 48-bit integer arithmetic.

[14] RPG also features the generation of random points within the unit disk.

[15] 192MB of main memory and adequate swap space did not suffice.

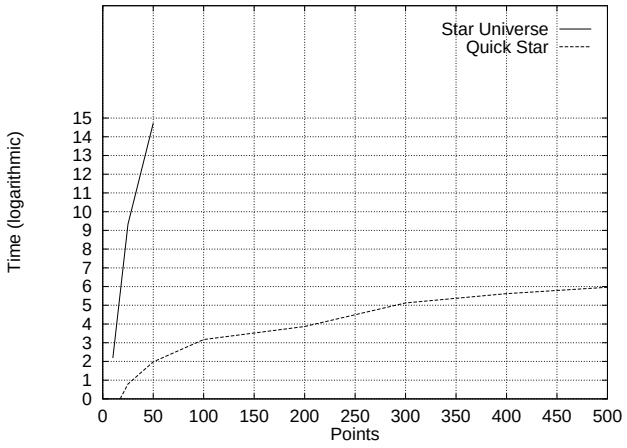

Figure 5: CPU-time consumption of the algorithms for the generation of star-shaped polygons.

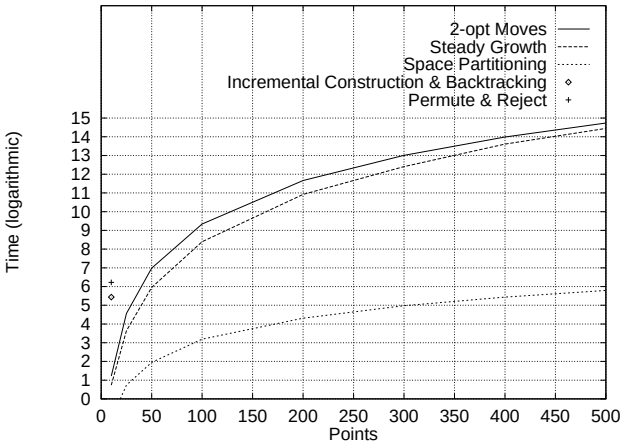

Figure 6: CPU-time consumption of the algorithms for the generation of simple polygons.

Two of the algorithms for the generation of simple polygons are not applicable to anything but extremely small point sets, cf. Fig. 6: 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. Clearly, those two algorithms are not suited for practical purposes. Note that using an $\mathcal{O}(n \log n)$ test for polygonal simplicity instead of our brute-force $\mathcal{O}(n^2)$ test would not improve the applicability of Permute & Reject at all.

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. Thus, from a performance point of view, Space Partitioning is the candidate of choice. Note that Quick Star and Space Partitioning consume about the same amount of CPU-time; i.e., Space Partitioning can be expected to be about as fast as sorting is if a somewhat fancy comparison function is used. Unfortunately, the CPU-time consumption of 2-opt Moves and Steady Growth may become prohibitively large if repeatedly applied to several thousands of points. (Memory consumption is no issue for any of these three methods, though.)

Clearly, the CPU-time consumption of 2-opt Moves highly depends on the number of 2-opt moves which it has to execute in order to transfer the original polygon into a simple polygon. Fig. 7 depicts the mean number of 2-opt moves executed by 2-opt Moves. Our tests indicate that a slightly super-linear number of 2-opt moves is needed, which results in a (super-) quadratic overall time complexity of 2-opt Moves. In fact, Fig. 6 seems to indicate a slightly super-quadratic growth rate of the CPU-time consumption of 2-opt Moves. We note that our tests did not witness any number of 2-opt moves which were anywhere close to the theoretical bound of $\mathcal{O}(n^3)$.

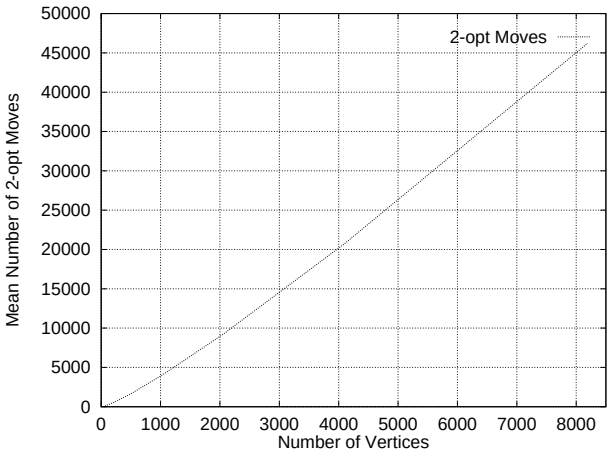

Figure 7: Mean number of 2-opt moves executed by 2-opt Moves.

3.2.2 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. 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. (All polygons were generated in CCW order, with the point with minimum x-coordinate as the first point.) 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.

The gigantic number of simple polygons on comparatively small sets of points also constitutes a practical problem when implementing any algorithm which is based on an enumeration of all polygons: integer overflows are very likely to occur[16], unless one switches to variable-length integer arithmetic. Also, the conventional random number generators (as contained in the standard libraries) can no longer be applied without modifications because they do not generate sufficiently large random numbers.

	Simple		Star-Shaped				
$\lvert\mathcal{S}_i\rvert$	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: Results for the numbers of simple and star-shaped polygons.

3.2.3 Quality Assessment

For each algorithm, we started with experimentally determining the ratio of the number of polygons generated and the total number of possible polygons.

For star-shaped polygons we have results for sets with 20 and 25 points. 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.

In the case of simple polygons, we tested 10 groups of sets with 10, 15 and 100 random points each. For sets with 10 or 15 points, we have results for 10,000 and 100,000 polygons generated on each set. For sets with 100 points we have results for 100,000 polygons generated. ((Due to lack of space, no test results for sets of 15 points are included in this paper; see [Aue96] for details.)

Among the methods for simple polygons, one method is significantly worse than the others: Incremental Construction & Backtracking. As can be seen in Fig. 8, 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. 9: 2-opt moves generates almost all polygons, Steady Growth lies around or above 90 percent, and Space Partitioning generates about between 80 and 90 percent of all possible polygons. In both tests the distribution of the polygons turned out to be highly non-uniform, though.

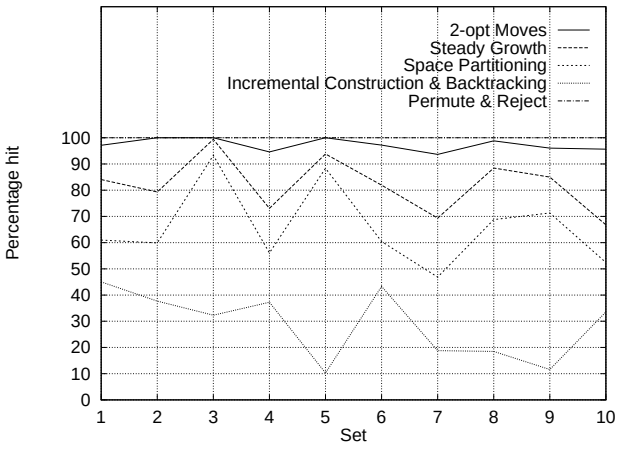

Figure 8: Results for 10,000 simple polygons on 10 points.

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

[16]Since we use the conventional integer arithmetic, we experienced integer overflows when running our implementation of the algorithm by Zhu et al. [ZSSM96] for 100 points.

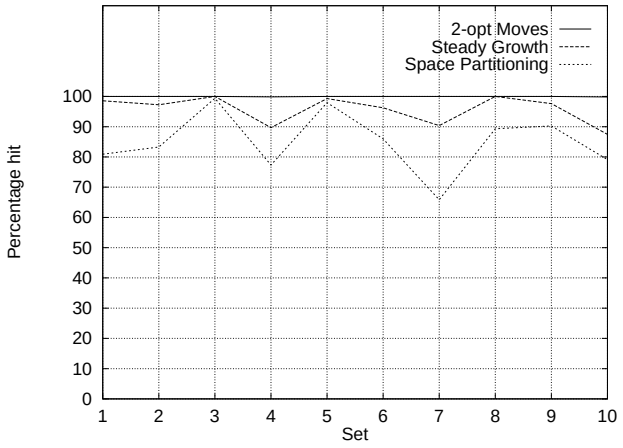

Figure 9: Results for 100,000 simple polygons on 10 points.

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 of simple polygons on 100 points. (We encountered sets with 20 points which already allowed more than three million simple polygons.) On one hand, our tests 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. (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.)

3.2.4 Characteristics of the Polygons Generated

We note, though, that the polygons generated by 2-opt Moves, Steady Growth and Space Partitioning seem to have different characteristics for larger sets of points. The polygons generated by Steady Growth and Space Partitioning tend to exhibit some "zigzagging" whereas the polygons generated by 2-opt Moves are much more pleasing from a visual point of view. This visual clue is confirmed by a plot of the distribution of the internal angles of the polygons generated. Fig. 10 shows the distribution of the internal angles averaged over 100,000 polygons generated on each of 10 sets of 100 points. (For the plot we subdivided the interval $[0, \ldots, 360]$ into 72 buckets of equal size, and then assigned the angles to their appropriate buckets; the plot shows how many percent of the angles ended up in an individual bucket.) It is clearly visible that Space Partitioning and, to a minor extent, Steady Growth generate polygons with quite a few interior angles close to 0 and 360 degrees.

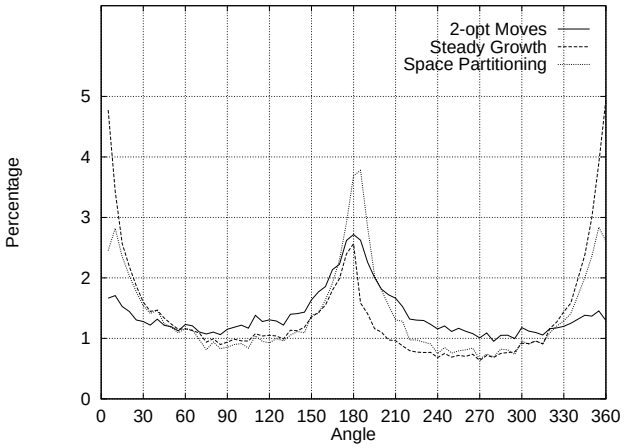

Figure 10: Distribution of the internal angles of the polygons.

In addition to the distribution of the angles we examined the sinuosities of the polygons generated. The "sinuosity" of a polygon has been used as a measure of the local complexity of a polygon, cf. [CI84]. Roughly, the sinuosity of a polygon gives the number of times the polygon's boundary alternates between spirals of opposite direction. Fig. 11 shows the distribution of the sinuosities. (Again, we examined 100,000 polygons generated on each of 10 sets of 100 points.) As it can be seen, about 17% of all polygons generated by 2-opt Moves had a sinuosity of 9, and about 55% of all polygons generated by 2-opt Moves had a sinuosity greater than or equal to 9. (About 1% actually had a sinuosity greater than or equal to 15, with 21 being the largest sinuosity encountered.) Steady Growth generates polygons with similar sinuosity values as 2-opt Moves. However, the sinuosity peak was at 8, and the maximal sinuosity encountered was 24. Space Partitioning seems to generate polygons with smaller sinuosity; only 10% of the polygons generated had a sinuosity greater than or equal to 9, with the maximal sinuosity being 18.

We also studied the asymptotic growth of the sinuosities as the number of vertices increases. Fig. 12 depicts the mean sinuosities for polygons ranging from 128 to 8192 vertices. (For each $7 \leq i \leq 13$, we randomly generated ten sets of 2^i points and computed one random polygon per point set.) Fig. 12 clearly

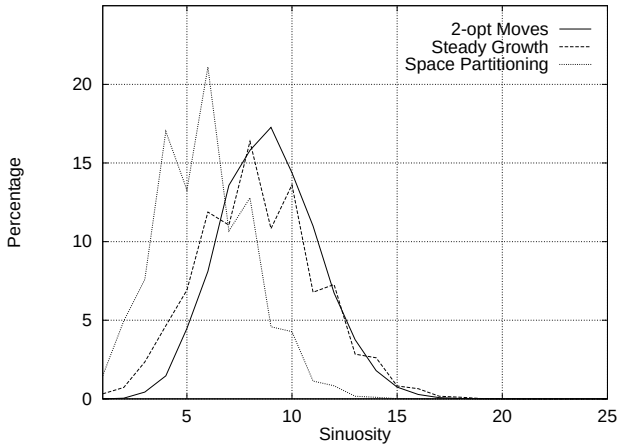

Figure 11: Distribution of the sinuosities of the polygons (100-gons).

shows that the sinuosities of the polygons generated do not seem to be bounded. Rather, the sinuosity of 2-opt Moves exhibits a nearly perfect linear growth, with an n-gon having a sinuosity of $0.079n$ on the average. As reflected by our extensive tests for sets of 100 points, the sinuosities of the polygons generated by Steady Growth and Space Partitioning are somewhat smaller on the average, but both exhibited (near-)linear growth rates, too. Thus, all three algorithms generate polygons of increasing complexity for increasing numbers of vertices, at least when taking the sinuosity as a measure of complexity.

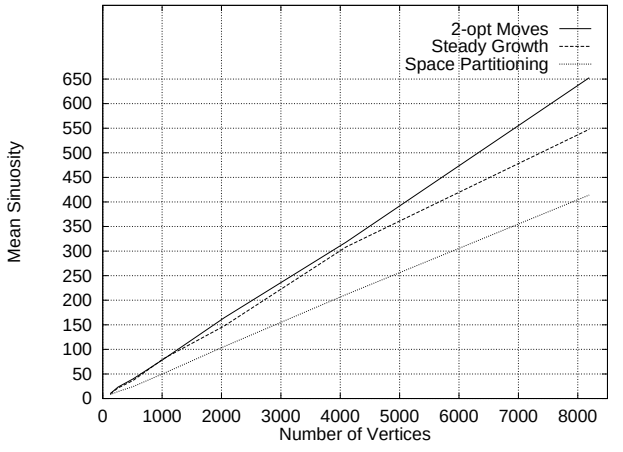

Figure 12: Asymptotic growth of the sinuosities.

We note that all polygons are quite distinct from a "smooth" polygon typically obtained by discretizing some free-form curve. In order to remedy this shortcoming RPG offers Smooth, which is a "smoothing" operation that takes as input any of the simple polygons generated by one of our heuristics, and produces a new simple polygon with twice the number of vertices. Basically, vertex v_i is replaced by the new vertices $\frac{v_{i-1}+3v_i}{4}$ and $\frac{v_{i+1}+3v_i}{4}$. (The simplicity of the resulting polygon is ensured by applying 2-opt Moves to it.) Of course, smoothing can be applied repeatedly in order to achieve any desired degree of smoothness.

It is no surprise that experimental tests of the distribution of the internal angles of a smoothed polygon showed one distinctive peak at 180 degrees, with the peak getting the higher the more often Smooth had been applied to a polygon. Our previous test results for the sinuosity also carry over quite naturally: a polygon with n vertices that was the result of k-fold smoothing can be expected to have a sinuosity corresponding to a polygon with $n/2^k$ vertices, where the actual sinuosity value again depends on the method used for generating the (non-smoothed) polygon.

In our opinion, excellent smooth polygons can be created by applying Smooth 2–4 times to the output of 2-opt Moves. We tried to back this claim by comparing our smoothed polygons to real-world polygons. However, despite of an inquiry on several mailing lists, we have not been able to obtain a statistically relevant number of real-world polygons.

Presumably, the polygons generated by 2-opt Moves come closest to complex polygons actually occurring in practice. Of course, we cannot really expect to get "real-world" polygons after subjecting our heuristics to input sets formed by points distributed uniformly at random within the unit cube. Unfortunately, generating real-world vertex sets seems to be nearly as hard as generating real-world polygons.

4 Conclusion

4.1 Summary

We presented five heuristics for the random generation of simple polygons. Three of these heuristics, namely 2-opt Moves, Steady Growth and Space Partitioning, are suited for practical purposes. However, for small point sets of, say, less than 100 points, we experienced a clear trade-off between the quality of the heuristic and the running time. Thus, when the CPU-consumption is not at a premium, one can afford to generate a large variety of complex 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. Smooth can be used as a postprocessor for any of these algorithms in order to generate realistic "smoothed" polygons.

Our experimental tests (and our personal experience with the practical use of RPG) indicate that the same trade-offs also hold true for the generation of random polygons on larger sets of points. If enough CPU-power is available then 2-opt Moves is likely to be the best heuristic. However, the class of simple polygons on large point sets is rich enough and the power of 2-opt Moves, Steady Growth and Space Partitioning is large enough that any of them can be expected to yield fairly good results. In particular, it is quite unlikely that a simple polygon will be generated repeatedly by any of these three heuristics.

We also ran experiments in order to assess a few key characteristics of the polygons generated, e.g., the sinuosities of the polygons: 2-opt Moves, Steady Growth, and Space Partitioning all generate polygons such that the growth rate of the sinuosity is linear in the number of vertices.

For the random generation of star-shaped polygons we presented a fast heuristic, Quick Star. It has a similar CPU-time consumption as Space Partitioning. Unfortunately, the enumeration algorithm Star Universe does not work for sets with more than, say, 50 points since it requires far too much main memory.

This paper and the RPG source code can be accessed via the WWW at the following URL: http://www.cosy.sbg.ac.at/~held/projects/rpg/rpg.html.

4.2 Open Problems

From a theoretical point of view, it remains an open problem to generate polygons on a given set of vertices uniformly at random. It is also natural to ask for tighter lower and upper bounds on the maximum number of star-shaped polygons on a given set of n points. And, if the upper bound were $o(n^4)$, can one enumerate all star-shaped polygons in $o(n^4)$ time?

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. Also, smarter ways for defining and measuring certain aspects of "randomness" for polygons (or similarly complex objects) would be very helpful.

From a practical point of view, however, it is not entirely clear what constitutes a good "random" polygon. A typical user may want to generate "random" polygons within some fuzzy subclass of polygons in order to best simulate the expected class of inputs for his application: e.g., generate random polygons which consist of two dominant "roughly convex" regions linked by a "roughly x-monotone" tunnel. And for a particular heuristic, a user might want to be able to classify the classes of polygons which are likely to be generated by the heuristic in some intuitive manner.

Evaluating the polygons generated turned out to be as challenging a task as generating them. Clearly, the sinuosity of a polygon is only one of a few complexity measures of a polygon. Also, the sinuosity serves only as a local measure of a polygon's complexity. A global measure that might be of interest is the polygon's link diameter, i.e., the minimum number of turns which any path (in the interior of the polygon) between two vertices has to make in the worst-case. An experimental study of the link diameter would allow to assess the "snakedness" of the polygons generated.

Another topic for future work is the random generation of multiply-connected planar areas: Given a set $\mathcal{S}$ of n points and an integer $k <= \frac{n}{3}$, generate k random polygons on $\mathcal{S}$ which bound a multiply-connected planar area. For practical applications it would also be of interest to generate simple random curves consisting of a more general class of elements, e.g., of lines and circular arcs.

Acknowledgments

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

References

[AH96] T. Auer and M. Held. Heuristics for the Generation of Random Polygons. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 38–44, Ottawa, Canada, Aug 1996. Carleton University Press.

[Aue96] T. Auer. Heuristics for the Generation of Random Polygons. Master's thesis, Computerwissenschaften, U. Salzburg, A–5020 Salzburg, Austria, June 1996.

[BS96] H. Bieri and P.-M. Schmidt. On the Permutations Generated by Rotational Sweeps of Planar Point Sets. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 179–184, Ottawa, Canada, Aug 1996. Carleton University Press.

[Cha91] B. Chazelle. Triangulating a Simple Polygon in Linear Time. *Discrete Comput. Geom.*, 6:485–524, 1991.

[CI84] B. Chazelle and J. Incerpi. Triangulation and Shape-Complexity. *ACM Trans. Graph.*, 3(2):135–152, Apr 1984.

[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.

[Knu69] D.E. Knuth. *The Art of Computer Programming, Vol. II: Seminumerical Algorithms.* Addison-Wesley, 1969.

[O'R94] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994. ISBN 0-521-44592-2.

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

[PS90] F.P. Preparata and M.I. Shamos. *Computational Geometry - An Introduction.* Springer-Verlag, third edition, Oct 1990. ISBN 3-540-96131-3.

[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.

[ZO95] T.C. Zhao and M. Overmars. Forms Library. *http://bragg.phys.uwm.edu/xforms*, 1995.

[ZSSM96] C. Zhu, G. Sundaram, J. Snoeyink, and J.S.B. Mitchell. Generating Random Polygons with Given Vertices. *Comput. Geom. Theory and Appl.*, 6(5):277–290, Sep 1996.