On Generating Polygons: Introducing the Salzburg Database*

Günther Eder[1], Martin Held[1], Steinþór Jasonarson[1], Philipp Mayer[1], and Peter Palfrader[1]

1 **Universität Salzburg, FB Computerwissenschaften, Salzburg, Austria,**
{geder,held,sjas,pmayer,palfrader}@cs.sbg.ac.at

Abstract

The Salzburg Database is a repository of polygonal areas of various classes and sizes, with and without holes. Positive weights are assigned to all edges of all polygons. We introduce this collection and briefly describe the generators that produced its polygons. The source codes for all generators as well as the polygons generated are publicly available.

1 Introduction

An important part of software development is testing the correctness and evaluating the performance of an algorithm implementation. Ideally, one would run the code 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. Subjecting the code to inputs of different characteristics is important since this may help to trigger different execution paths. Similarly, a large range of input sizes is needed to obtain insights in the actual runtime and memory consumption. This allows for comparing different implementations in a meaningful way.

Our goal with the Salzburg Database is to provide a repository of data for such testing purposes. The initial content of the Salzburg Database is purely polygonal, containing simply-connected and multiply-connected polygonal areas in two dimensions.

Every polygon has positive weights assigned to its edges. These weights can be used to test codes that operate on weighted polygonal input, such as for computing weighted straight skeletons. The file format is extensible, so we can also add vertex-weights and other information such as edge or vertex colorings in the future.

We note that this is work in progress. In particular, we are still evaluating the generators and the characteristics of the polygons generated by them. Hence, we expect to see some fine tuning of the generators in the near future. In the sequel we describe the database and its generators.

2 Generators

Generating simple polygons is not a new problem. For convex and x-monotone polygons, Zhu et al. [9] propose a solution to generate them uniformly at random. Tomás and Bajuelos [7] introduce a quadratic-time algorithm to generate random polygons on a grid. Dailey and Whitfield [3] describe a heuristic that takes $\mathcal{O}(n \log n)$ time to compute a simple polygon. They start from a randomly chosen triangle followed by repetitive edge subdivision. Sedhu et al. [5] introduce a different heuristic, which constructs a random polygon starting from the convex hull of a given point set. They randomly select a vertex inside the hull and add

* Work supported by Austrian Science Fund (FWF): Grants ORD 53-VO and P31013-N31.

36th European Workshop on Computational Geometry, Würzburg, Germany, March 16–18, 2020.
This is an extended abstract of a presentation given at EuroCG'20. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear eventually in more final form at a conference with formal proceedings and/or in a journal.

it to the polygon while maintaining the simplicity using visibility checks. Later, Sedhu et al. [6] introduce a different approach that uses the convex layers of a given point set and constructs a simple polygon in $\mathcal{O}(n \log n)$ time.

As this brief overview of the literature demonstrates, clearly some research has been devoted to this topic. Here, we present our generators and their actual implementations, some of which, like RPG (Section 2.4) or FPG (Section 2.1), implement algorithms from, or inspired by, literature.

2.1 Triangulation Perturbation

Our implementation FPG is motivated by an approach originally proposed by O'Rourke and Virmani [4]: They start with a regular polygon $\mathcal{P}$ and then translate its vertices while maintaining the polygon's simplicity. A direction and speed are chosen at random and assigned to each vertex of $\mathcal{P}$. Then, the vertices of $\mathcal{P}$ are processed consecutively. A single vertex is moved one "time unit" as long as $\mathcal{P}$ remains simple, otherwise that move is omitted and a new random velocity is chosen for the next round. O'Rourke and Virmani [4] suggest to use several hundred translations per vertex.

As vertices can also move in an outward direction, a domain is defined which has to contain $\mathcal{P}$. We use a large rectangle to limit the outward movement of the vertices.

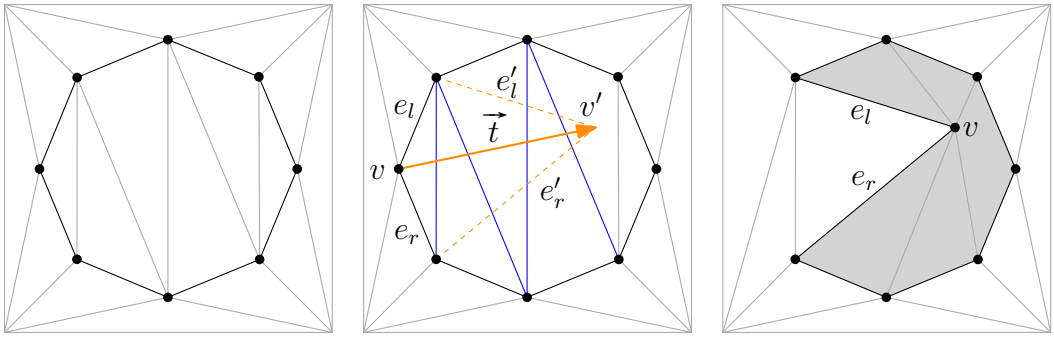

Figure 1 (a) Triangulation of the start polygon and its domain; (b) Translation of vertex v by the vector $\vec{t}$; (c) The polygon after the translation.

Maintaining the simplicity of $\mathcal{P}$ during the vertex translations can be an expensive task if carried out naively. We utilize a triangulation of the interior and the exterior of $\mathcal{P}$ to simplify intersection tests while moving a polygon vertex; cf. Figure 1a. Let v denote a boundary vertex of $\mathcal{P}$ that we want to translate and let e_l and e_r denote its two incident edges. In practice, a randomly chosen translation vector $\vec{t}$ tends to violate the simplicity of $\mathcal{P}$, with high probability, which leads to a bad performance. Therefore, we choose a random direction for $\vec{t}$ first. Then, the length of $\vec{t}$ is generated from a normal distribution using parameters suitable to the local environment around v, in the chosen direction. Experiments show that such an approach for choosing translation vectors produces only few invalid translations.

After translating v by $\vec{t}$, we obtain v' and the edges e'_l and e'_r, respectively. Our intersection test involves checking all triangles pierced by e'_l or e'_r. In case all triangle edges intersected by e'_l and e'_r are interior or exterior diagonals, we change v into v' in $\mathcal{P}$. Additionally, we may have to modify the triangulation by checking the triangles intersected by the modified edges as well as the triangles incident at v. If we cross a polygon edge,

we reject $\vec{t}$ as translation vector and restart the process. See an example of this process illustrated in Figures 1b and 1c.

FPG starts from a regular polygon where a triangulation, in- and outside, is trivially obtained. To speed up the generation of large polygons, instead of starting with a large regular polygon, FPG can start with a smaller one, and then "grow" this polygon by repeatedly splitting random edges. The additional vertex introduced by the split is then translated to avoid collinearities.

If we pick edges uniformly at random, we see clusters of many short edges and a few very long edges. This presumably is due to the fact that areas with short edges are more likely to get extra vertices than areas of the same size which contain (fewer) long edges; cf. Figure 2. To avoid this clustering, we instead pick edges randomly weighted by their length.

Furthermore, FPG is capable of generating polygons with holes. Since $\mathcal{P}$ is regular at the beginning, we can trivially place regular holes inside $\mathcal{P}$ as well. The described process works also for this setting, as the intersection tests hinge on the triangulation. In Figure 3 we illustrate the evolution of a polygon computed by FPG, the polygon has 10 vertices, with a triangular hole formed by three additional vertices. The first two images in Figure 4 are the result of FPG using edge-subdivision; the second image depicts a polygon with holes.

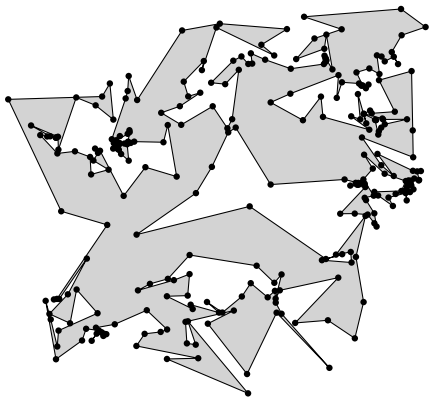

Figure 2 Polygon exhibiting clustering due to the selection of edges uniformly at random in the subdivision step.

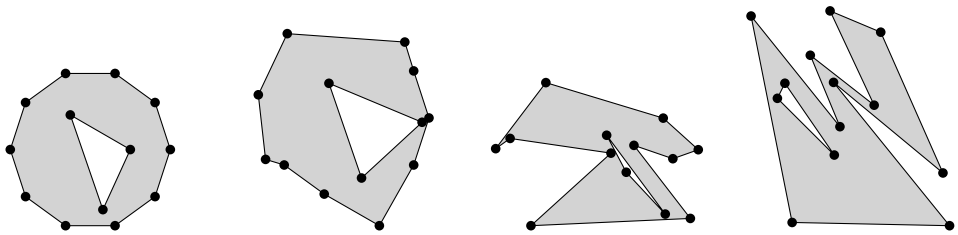

Figure 3 Polygon generated by FPG after 1, 8, 50, and 500 iterations without edge-subdivision.

2.2 Combining Line Sweep and 2-Opt Moves

Our generator SPG constructs a simple polygon $\mathcal{P}$ on a given point set S in the plane. (Such a point set can be generated randomly or specified by a user.) Initially, SPG creates a polygon by choosing a random permutation of the input vertices. This start-polygon contains, with high probability, self-intersections. Therefore, a line sweep is applied to identify intersecting pairs of edges, followed by local modifications which remove these intersections.

To identify pairs of edges that intersect we use the classic Bentley-Ottmann algorithm [2]. We sweep from left to right, thereby maintaining a sorted set of edges that intersect the sweep-line. The input vertices comprise the event points of the line sweep. During the sweep, at vertex v_i, we have to modify the sweep-line status by removing and/or adding the edges

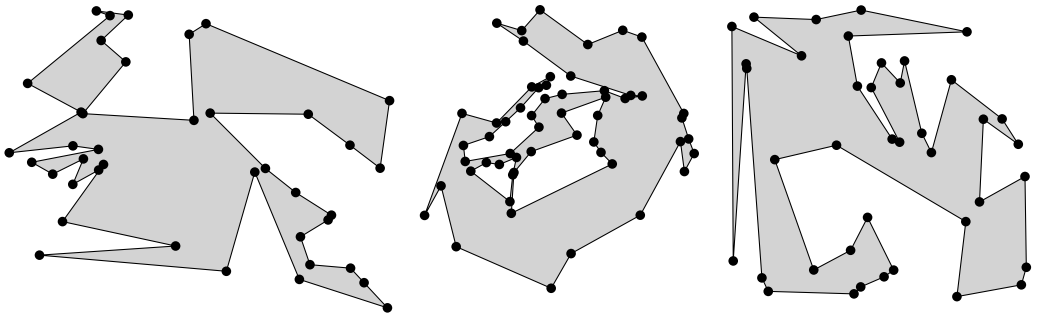

Figure 4 Left-to-right: A polygon and a polygon with holes computed by FPG, and a polygon generated by SPG.

incident at v_i. Additionally, at every event point, we have to verify that any newly added edge is not intersecting its neighbors in the status. In case a pair of edges does intersect, we have to resolve that intersection before we carry on with the sweep.

We resolve intersections by applying so-called *2-opt* moves. A 2-opt move replaces the edges $e_1 = \overline{v_1v_2}$ and $e_2 = \overline{v_3v_4}$ by the edges $e'_1 = \overline{v_1v_3}, e'_2 = \overline{v_2v_4}$. (Note that the polygon boundary becomes disconnected if the 2-opt move connects the wrong vertex pairs.) As we apply 2-opt moves during the line sweep to resolve intersections, we may introduce new intersections. However, a key property of the 2-opt move is that it decreases the length of the polygon (if not all points are collinear). This guarantees that we will eventually arrive at a polygon that is simple if we apply 2-opt moves repeatedly to resolve intersections. A result by van Leeuwen and Schoone [8] tells us that we need at most $\mathcal{O}(n^3)$ 2-opt moves.

We implemented and tested three variants of the line sweep. They differ mainly in how they proceed after finding and resolving an intersection: (a) After a 2-opt move is carried out, we simply continue with the line sweep. After arriving at the right-most vertex we restart the line sweep at the left-most vertex. The sweep is repeated until all intersections are resolved. (b) After a 2-opt move, we test and resolve all intersections at the current sweep-line position, before carrying on. Again, at the right-most vertex we restart until all intersections are resolved. (c) After a 2-opt move, we reverse the sweep direction to deal with possibly new edge intersections. We resume our rightwards sweep at the left-most vertex affected by the 2-opt move. The last image in Figure 4 was generated by SPG on a point set of 40 vertices using sweep-variant (a).

Note that collinear edges need special care because a 2-opt move will not always result in a shortening of the perimeter of the polygon. If intersecting collinear edges are detected, then we remove these edges and sort the respective collinear vertices. Then, we connect the vertices by edges in consecutive order, i.e., form a chain of non-overlapping collinear edges. This guarantees that the perimeter of the polygon decreases also in the case of collinear vertices.

2.3 SRPG

SRPG generates simply-connected and multiply-connected polygonal areas by means of a regular grid that consists of square cells. Given two integer values, a and b, SRPG generates a grid of size a times b. By default SRPG then generates orthogonal polygons on this grid. An additional parameter p, between zero and one, leads to a smaller or larger number of vertices in the produced polygon. SRPG is able to produce octagonal polygons by cutting off

corners with $\pm 45°$ diagonals during the construction. Cutting corners repeatedly, without the diagonal restriction, yields an approximation of a smooth free-form curve. Additionally, SRPG can apply perturbations in order to generate polygons with axes-parallel edges whose vertices do not lie on a grid, or to generate polygons whose edges (in general) are not parallel to the coordinate axes. See Figure 5 for some sample polygons.

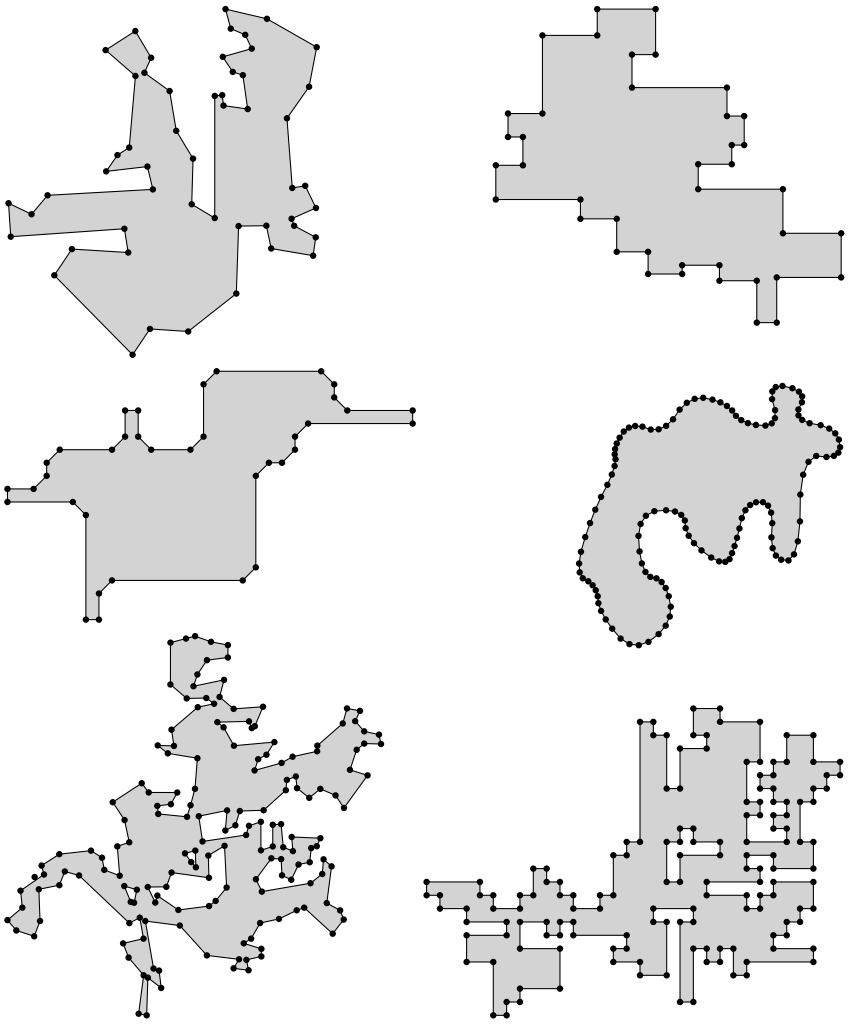

Figure 5 Samples of a random, an orthogonal, an octagonal, and a smoothed polygon generated by SRPG, as well as a random and a grid-aligned orthogonal polygon with holes.

2.4 RPG

Auer and Held [1] first described RPG more than twenty years ago. RPG supports various heuristics to generate "random" polygons for a given set of vertices. In particular, it is able to produce star-shaped polygons uniformly at random. Furthermore, it generates x-monotone polygons uniformly at random, based on the algorithm by Zhu et al. [9]. We have resurrected this code and updated it to compile on modern platforms, thus meeting requests voiced by several colleagues. A recent extension of RPG also supports the generation of polygons with holes. See Figure 6 for examples of some polygons generated by RPG.

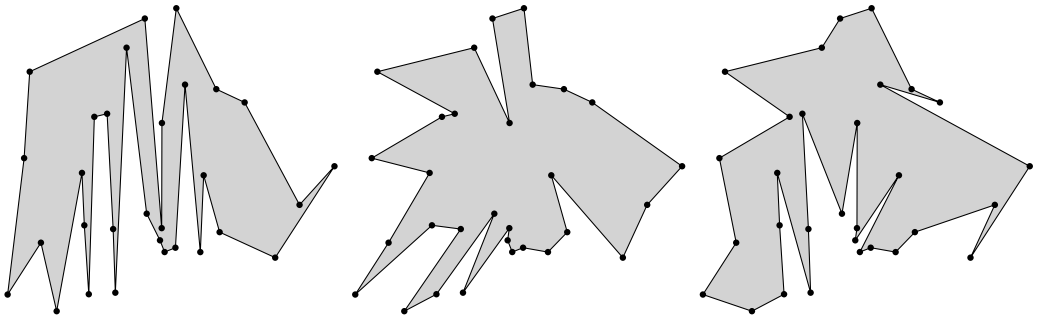

Figure 6 In left-to-right order, an x-monotone, a star-shaped, and a simple polygon computed by RPG on 30 vertices.

2.5 Additional Generators

Our repository also contains codes to produce well-known polygons such as the Koch snowflake (also in a nested variant), the Sierpinski curve, and closed variants of the Hilbert and Lebesgue curves; see Figure 7.

3 Salzburg Database

The Salzburg Database is available at `https://sbgdb.cs.sbg.ac.at/`. Since this is work-in-progress, we expect to add additional data-sets and generators in the near future. The database can be used freely and is provided via direct download or git.

Currently, all our generators are written in C++ or plain C. However, we are not averse to adding code written in other languages such as Python. All source code available on GitHub (`https://github.com/cgalab`) and can be used freely under the GPL(v3) license.

We conclude this survey of the Salzburg Database with a call for participation. If you have "interesting" polygons or data-sets you like to have included then, please, send them to us. You are also welcome to to contact us if you have an interest in a specific class of polygons that is missing.

References

1 T. Auer and M. Held. Heuristics for the Generation of Random Polygons. In *Proceedings of the 8th Canadian Conference on Computational Geometry (CCCG)*, pages 38–44, 1996.

2 J. L. Bentley and T. A. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.

3 D. Dailey and D. Whitfield. Constructing Random Polygons. In *Proceedings of the 9th ACM SIG-Information Technology Education Conference (SIGITE)*, pages 119–124, 2008.

4 J. O'Rourke and M. Virmani. Generating random polygons. Technical report, Smith College, Northampton, MA 01063, USA, 1991.

5 S. Sadhu, S. Hazarika, K. Jain, S. Basu, and T. De. GRP_CH Heuristic for Generating Random Simple Polygon. In *International Workshop on Combinatorial Algorithms (IWOCA)*, 2012.

6 S. Sadhu, N. Kumar, and B. Kumar. Random Polygon Generation through Convex Layers. *Procedia Technology*, 10:356–364, 2013.

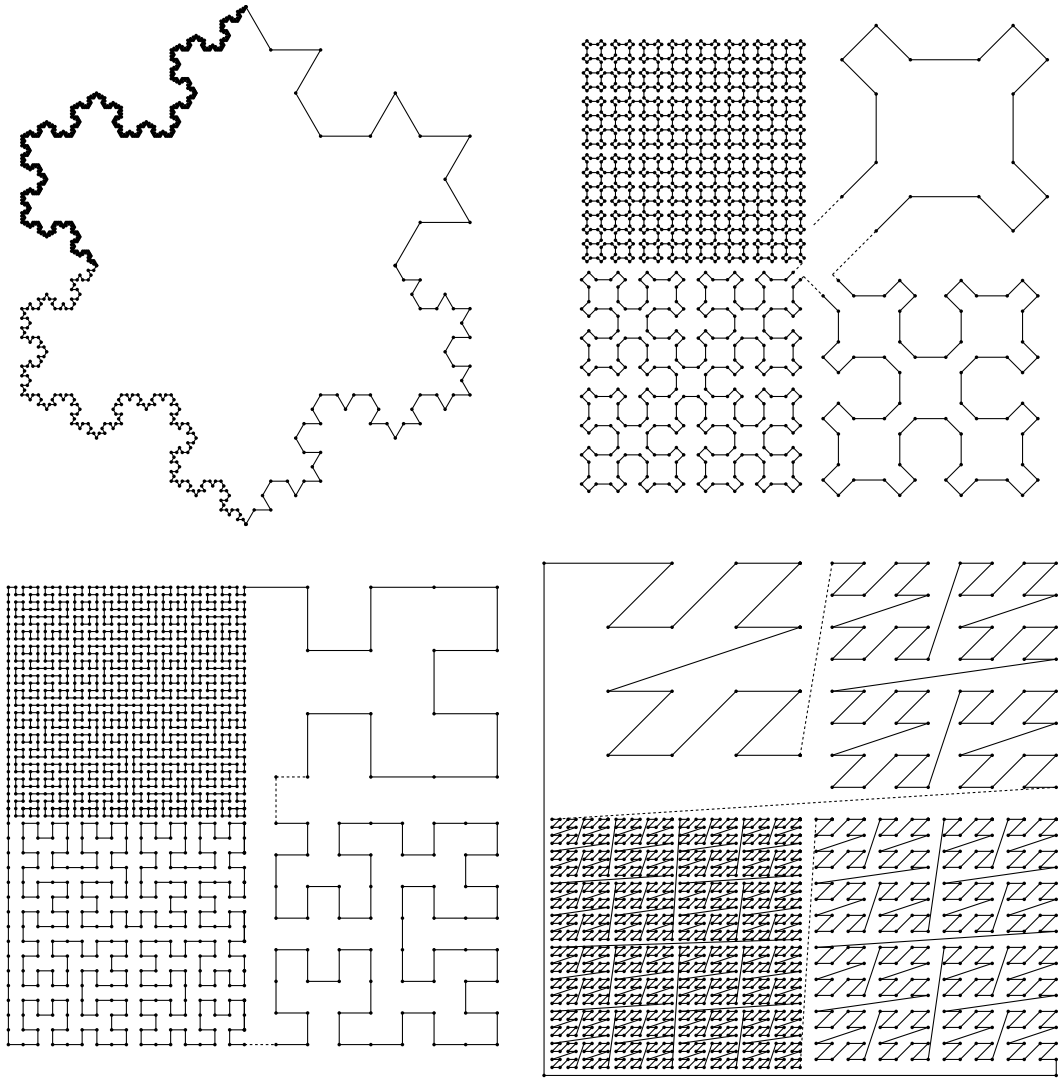

Figure 7 The curves of Koch, Sierpinski, Hilbert, and Lebesgue, in reading order. Each figure is partitioned into four quadrants which portions of the curve at different orders.

7 A. Tomás and A. Bajuelos. Quadratic-Time Linear-Space Algorithms for Generating Orthogonal Polygons with a given Number of Vertices. In *Computational Science and Its Applications (ICCSA)*, pages 117–126, 2004.

8 J. van Leeuwen and A. A. Schoone. Untangling a Travelling Salesman Tour in the Plane. In J. Mühlbacher, editor, *Proc. 7th Conference Graph-theoretic Concepts in Computer Science (WG'81)*, pages 87–98, 1982.

9 C. Zhu, G. Sundaram, J. Snoeyink, and J. Mitchell. Generating Random Polygons with Given Vertices. *Computational Geometry: Theory and Applications*, 6(5):277–290, 1996.