Experimental Evaluation of Straight Skeleton Implementations Based on Exact Arithmetic*

Günther Eder[1], Martin Held[1], and Peter Palfrader[1]

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

Abstract

We present C++ implementations of two algorithms for computing straight skeletons in the plane, based on exact arithmetic. One code, named Surfer2, can handle multiplicatively weighted planar straight-line graphs (PSLGs) while our second code, Monos, is specifically targeted at monotone polygons. Both codes are available on GitHub. We sketch implementational and engineering details and discuss the results of an extensive performance evaluation in which we compared Surfer2 and Monos to the straight-skeleton package included in Cgal. Our tests provide ample evidence that both implementations can be expected to be faster and to consume significantly less memory than the Cgal code.

1 Introduction

Straight skeletons were introduced to computational geometry by Aichholzer et al. [2]. Suppose that the edges of a simple polygon P move inwards with unit speed in a self-parallel manner, thus generating mitered offsets inside of P. Then the *(unweighted) straight skeleton* of P is the geometric graph whose edges are given by the traces of the vertices of the shrinking mitered offset curves of P. The process of simulating the shrinking offsets is called *wavefront propagation*. In the presence of multiplicative weights, wavefront edges no longer move at unit speed. Rather, every edge moves at its own constant speed; see Figure 1. Straight skeletons are known to have applications in diverse fields, with the modeling of roof-like structures being one of the more prominent ones [14, 10, 11]. We refer to Huber [12] for a detailed discussion of typical applications.

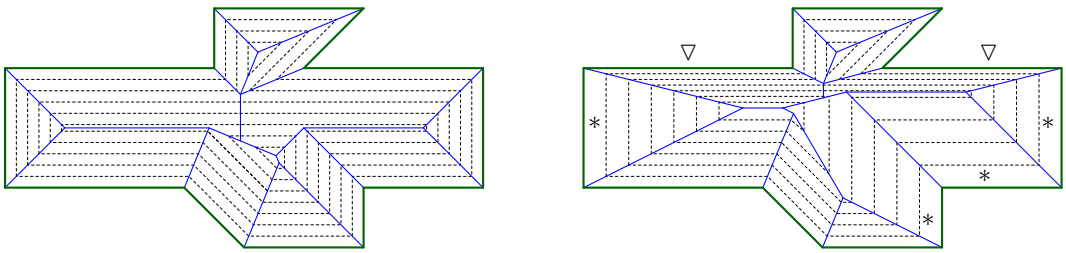

Figure 1 Left: The (unweighted) straight skeleton (in blue) plus a family of wavefronts (dashed) for the green polygon. Right: The weighted straight skeleton for the case that edges marked with $*$ have twice the weight and edges marked with $\triangledown$ have half the weight of the unmarked edges.

The straight-skeleton algorithms with the best worst-case bounds are due to Eppstein and Erickson [9] and Vigneron et al. [6, 17]. These algorithms seem difficult to implement. Indeed, progress on implementations has been rather limited so far. The first comprehensive code for computing straight skeletons was implemented by Cacciola [5] and is shipped with

* 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.

CGAL [16]. It handles polygons with holes as input. The straight-skeleton code STALGO by Huber and Held [13] handles PSLGs as input and runs in $O(n \log n)$ time and $O(n)$ space in practice. (A PSLG is an embedding of a planar graph such that all edges are straight-line segments which do not intersect pairwise except at common end-points.) However, it would be difficult to extend to weighted skeletons [7].

2 Implementations

Monotone Polygons. Biedl et al. [4] describe an $\mathcal{O}(n \log n)$ time algorithm to compute the straight skeleton of a simple n-vertex x-monotone polygon $\mathcal{P}$. Their algorithm consists of two steps: (1) The polygon $\mathcal{P}$ is split into an upper and lower monotone chain, and the straight skeleton of each chain is computed individually by means of a classical wavefront propagation. (2) The final straight skeleton $\mathcal{S}(\mathcal{P})$ is obtained by merging these two straight skeletons.

Weighted PSLGs. Aichholzer and Aurenhammer [1] describe an algorithm for computing the straight skeleton of general PSLGs in the plane. It carries over to multiplicatively weighted input in a natural way, provided that all weights are positive. Their approach constructs the straight skeleton by simulating a wavefront propagation. As the wavefront sweeps the plane, they maintain a kinetic triangulation of that part of the plane which has not yet been swept. This triangulation is obtained by triangulating the area inside the convex hull of all wavefronts. Furthermore, all edges of the convex hull are linked with a dummy vertex at infinity.

The area of each triangle of this kinetic triangulation changes over time as its vertices, which are vertices of the wavefront, move along angular (straight-line) bisectors of the input edges. Every change in the topology of the wavefront is witnessed by a triangle collapse. (But not all triangle collapses correspond to changes of the wavefront topology.) We refer to Palfrader et al. [15] for more details of this algorithm.

Codes. Our implementations, MONOS and SURFER2, of these two algorithms use CGAL's `Exact_predicates_exact_constructions_kernel_with_sqrt` algebraic kernel, which is backed by CORE's `Core::Expr` exact number type. (SURFER2 can also be run with standard IEEE 754 arithmetic.) Both source codes are provided on GitHub and can be used freely under the GPL(v3) license: `https://github.com/cgalab/monos` and `https://github.com/cgalab/surfer2`.

3 Engineering Aspects

Careful algorithm engineering was applied to both MONOS and SURFER2. Due to lack of space we only sketch a few of our engineering considerations. For instance, a major computational task to be carried out by MONOS during the merge step are intersection tests and intersection computations between bisectors and skeleton arcs. Initially, we applied CGAL's `do_intersect` and `intersection` rather naïvely to an arc segment (seen as a straight-line segment) and a bisector. However, explicitly deciding whether the end-points of an arc segment lie on different sides of the supporting line of a merge bisector is sufficient to decide whether an intersection occurs. Once we know that an intersection occurs then we apply CGAL's `intersection` routine to the supporting lines of the arc and the bisector. Tests showed average runtime savings of about 9 % when using the latter method. Similarly,

switching from C++'s `std::set` to a self-developed binary min-heap resulted in an average performance gain of 5 %.

While the actual collapse time of a triangle of the kinetic triangulation is one of the roots of a quadratic polynomial, solving quadratics is not always necessary. Avoiding root finding for quadratic polynomials will increase accuracy when working with limited-precision data types, and it will result in less complex expression trees when working with exact numbers as provided by CORE's `CORE::Expr`. In particular, it will avoid the computation of another square root. Hence, SURFER2 tries to employ geometric knowledge derived from local combinatoric properties as much as this is possible. For instance, consider a triangle with exactly one incident wavefront edge. Its collapse can correspond to an edge event, split event or flip event but we can determine each such event without computing the roots of a determinant: The times of split and flip events can be found by considering the distance between the vertex opposite the wavefront edge to the supporting line of the wavefront edge. This distance is linear in time and when it passes through zero, we either have a flip or split event as the vertex comes to lie on the supporting line of the wavefront edge.

The use of CORE's `CORE::Expr` makes it easy for SURFER2 to know which events happen simultaneously. The significant price to be paid is that comparisons are no longer unit-cost. Hence, SURFER2 attempts to reduce the number of actual comparisons of event times where possible. E.g., if a closed loop partitions the plane into two connected components then the straight skeleton within one component is entirely independent of the straight skeleton within the other component, thus allowing SURFER2 to avoid comparing event times if the events happen within different components.

4 Experimental Results

Setup. All runtime tests were carried out on a 2015 Intel Core i7-6700 CPU. For most of our tests, memory consumption was limited to 10 GiB. The codes were compiled with `clang++`, version 7.0.1, against CGAL 5.0 except where stated otherwise.

The default setting for CGAL's straight-skeleton package [5] is to use the exact predicates but inexact constructions kernel that ships with CGAL. The use of this kernel ensures that the straight skeleton computed is combinatorially correct, even if the locations of the nodes need not be correct. CGAL's straight skeleton package can also be run with the exact predicates and exact constructions kernel. However this causes the runtimes to increase by a factor of roughly 100. Our codes, MONOS and SURFER2, use the CGAL exact predicates and exact constructions with square-root kernel by default, as we construct wavefront vertices with velocities, and these computations involve square roots.

We tested both CGAL and SURFER2 on many different classes of polygons. Our test data consists of real-world (multiply-connected) polygons as well as of synthetic data generated by RPG [3] and similar tools provided by the Salzburg Database [8]. For a polygon that has holes we used CGAL's straight-skeleton code that supports holes. Otherwise we used the implementation which only supports simple polygons.

Additionally, we tested both of our codes, MONOS and SURFER2, with large monotone polygons, for up to 10^6 vertices. (We did not run CGAL on these inputs due to memory constraints.) Given the lack of a sufficiently large number of monotone real-world inputs, we used RPG [3] to automatically generate thousands of monotone input polygons. We also ran SURFER2 on hundreds of real-world PSLGs. Most of our data came from GIS sources and represents road and river networks, contour lines and the like. The runtimes on those inputs are quite comparable to the runtimes for polygons. That is, the test results presented

here are also representative for SURFER2's performance on real-world data.

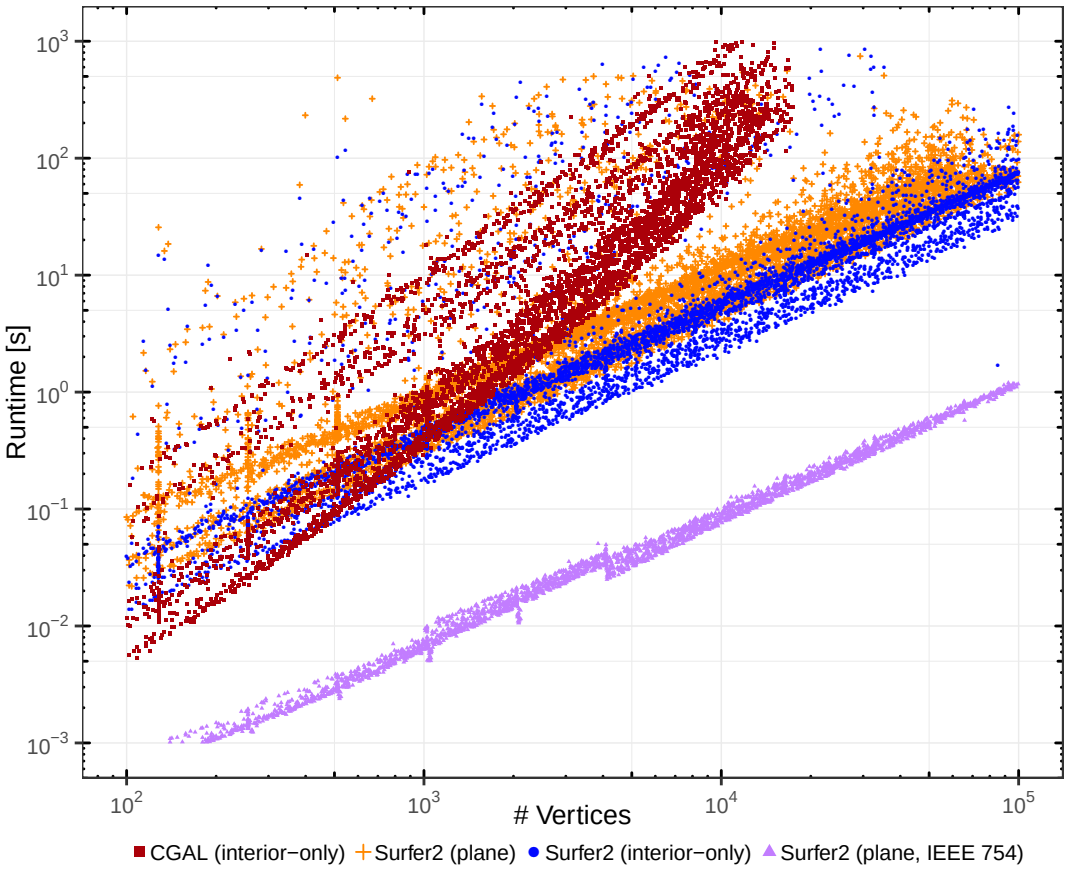

Figure 2 Runtimes of CGAL's code and different variants of SURFER2.

Runtimes and memory consumption. While CGAL's code computes the straight skeleton either in the interior or the exterior of a polygon, SURFER2 can do both in one run because it treats a polygon as a PSLG. But SURFER2 can also be restricted to just the interior or the exterior of a polygon. Hence, for the plot in Figure 2 we ran SURFER2 twice, once applied to the entire plane and once for just the interiors of the polygons that were also handled by CGAL. Our tests make it evident that SURFER2 is significantly faster than CGAL for a large fraction of the inputs. In particular, its runtime seems to exhibit an $n \log n$ growth, compared to the clearly quadratic increase of the runtime of CGAL's code. Figure 3 shows that SURFER2's memory consumption grows (mostly) linearly while CGAL's code requires a clearly quadratic amount of memory.

The results for CGAL's straight skeleton package were to be expected because (at least back in 2010) it computed potential split events for each pair of reflex vertex and wavefront edge [12, Section 2.5.4]. Indeed, tests carried out in 2010 indicated that it requires $\mathcal{O}(n^2 \log n)$ time and $\Theta(n^2)$ space for n-vertex polygons, as discussed in [13]. Our test results suggest that the same algorithm and implementation are applied in the current CGAL 5.0, which we used for our tests. The theoretical upper bound on the runtime complexity of SURFER2 is given by $O(n^3 \log n)$. Thus, its very decent practical performance is noteworthy.

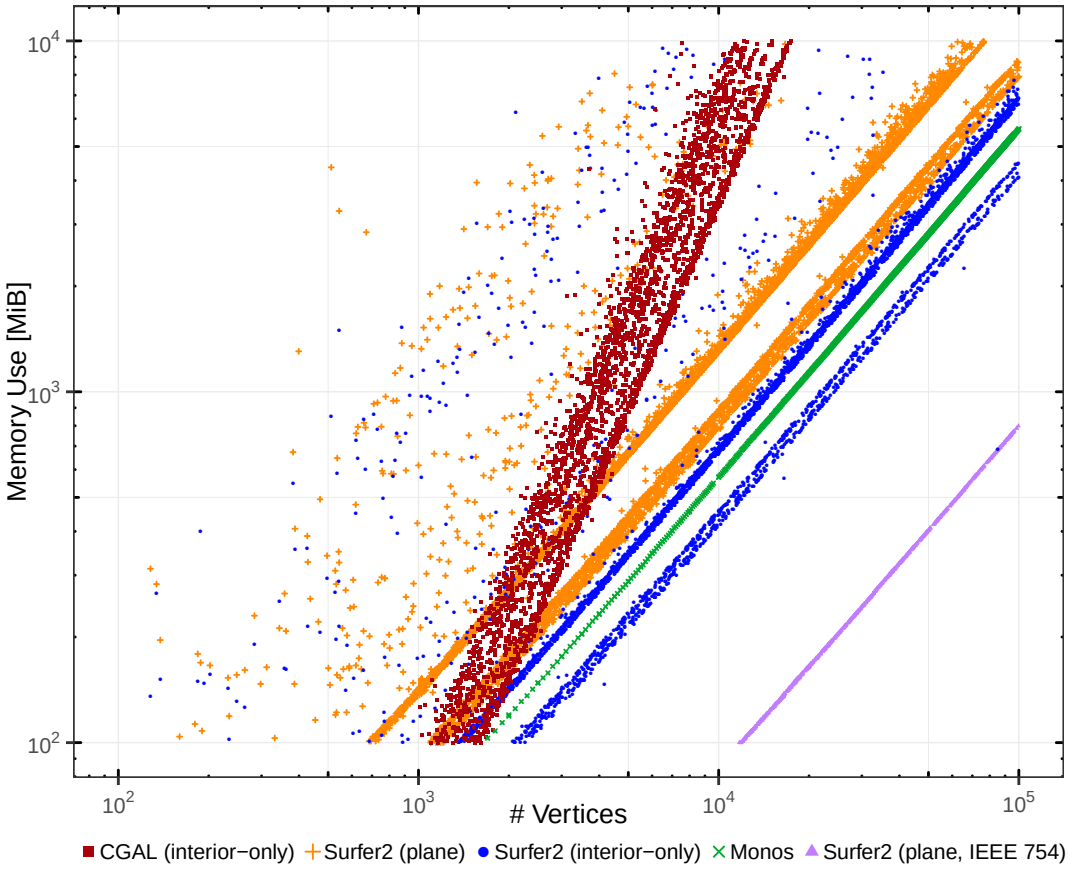

Figure 3 Memory use of CGAL, SURFER2 variants, and MONOS.

To see the runtime and memory characteristics of the practical costs of using `CORE::Expr`, we also ran SURFER2 with IEEE 754 double as a number type; cf. Figures 2 and 3.

The $\mathcal{O}(n \log n)$ bound for the complexity of MONOS is apparent in Figure 4, left. Given that MONOS is a special-purpose code designed specifically for handling monotone polygons, it had to be expected that it outperforms SURFER2 consistently.

Dependence on input characteristics. There are outliers both in the runtime as well as in the memory consumption of SURFER2 which are clearly visible in Figures 2 and 3. (To a lesser extent this noise is visible for CGAL, too.) Given the fact that such outliers do not show up for the IEEE 754-based version of SURFER2, there is reason to assume that this behavior is not intrinsic to SURFER2's algorithm but that it has its roots in the use of exact arithmetic. To probe this issue further we investigated which input classes trigger these outliers. Figure 5 shows the runtimes of both codes for three different input classes.

The class of input that was most time-consuming to handle for all implementations were our random octagonal polygons. All polygons out of this group have interior angles which are multiples of 45°. We were surprised to see that orthogonal polygons need not be troublesome per se. The key difference between both groups of polygons is given by the fact that all octagonal polygons have their vertices on an integer grid while the vertices of the orthogonal polygons are (random) real numbers. Hence, for the octagonal polygons many events tend to happen at exactly the same time when opposite edges become incident in different parts

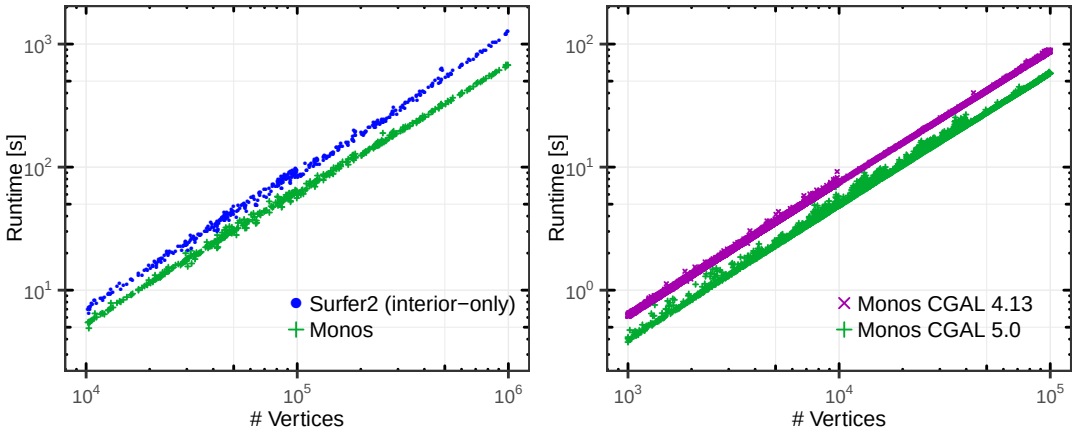

Figure 4 Left: Runtime of MONOS v. SURFER2 on large monotone inputs.
Right: Runtime of MONOS with CGAL 4 v. CGAL 5.

of the polygon. It is apparent that the proper time-wise ordering of these events incurs a significant cost if `CORE::Expr` is used. The presence of parallel edges does not automatically increase the runtime of SURFER2, though. The likely explanation is that many events are caused by opposite wavefront edges that become incident. In such a scenario, SURFER2 constructs a so-called infinitely-fast vertex. These vertices are easy to sort because they are handled immediately. For comparison purposes we ran the code on random simple polygons as a third class of input, with random vertex coordinates. Having any kind of co-temporal event is highly unlikely for those inputs, as are infinitely-fast vertices.

The excessive amounts of time consumed by ordering simultaneous events became even more apparent when we studied the impact of multiplicative weights on SURFER2's runtime. As expected, a difference in the timings for weighted and unweighted random polygons was hardly noticeable. For our randomly weighted octagonal polygons we expected reduced runtimes and fewer outliers even when using exact arithmetic, due to very few truly simultaneous events. And, indeed, this was confirmed impressively by our tests; see Figure 5.

Experiences with CGAL. While running our tests, we also compared CGAL versions 4.13 and 5.0, as the latter was released only recently. We witnessed an improvement in the performance of our codes for the newer version; see Figure 4, right.

References

1 O. Aichholzer and F. Aurenhammer. Straight Skeletons for General Polygonal Figures in the Plane. In *Voronoi's Impact on Modern Sciences II*, volume 21, pages 7–21. Institute of Mathematics of the National Academy of Sciences of Ukraine, 1998.

2 O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner. A Novel Type of Skeleton for Polygons. *Journal of Universal Computer Science*, 1(12):752–761, 1995.

3 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.

4 T. Biedl, M. Held, S. Huber, D. Kaaser, and P. Palfrader. A Simple Algorithm for Computing Positively Weighted Straight Skeletons of Monotone Polygons. *Information Processing Letters*, 115(2):243–247, 2015.

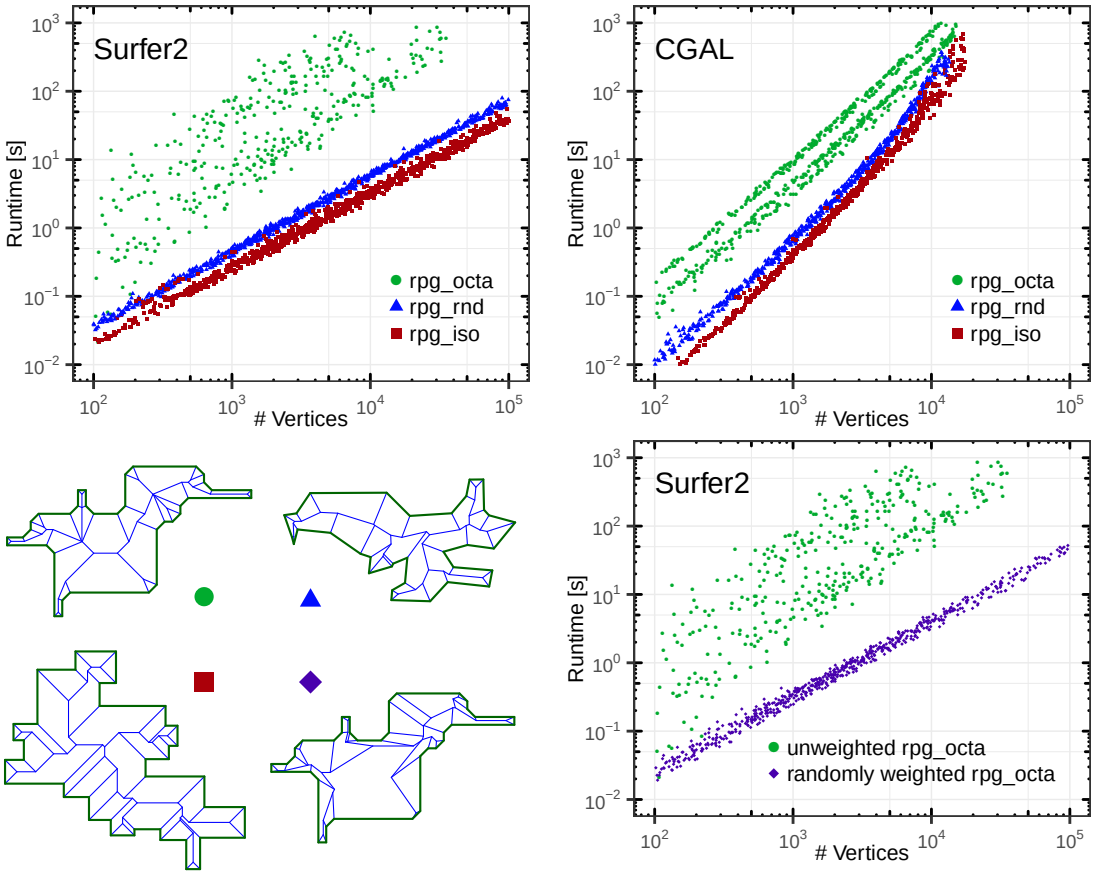

Figure 5 Top: The effect of different input classes on the runtime of SURFER2 and on CGAL. Bottom-left: Samples for different input classes (left to right, top to bottom): Octagonal input (on integer grid), random polygon, orthogonal polygon not on integer grid, and weighted octagonal input. Bottom-right: SURFER2 runtimes for unweighted and randomly weighted octagonal input.

5 F. Cacciola. 2D straight skeleton and polygon offsetting. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.0 edition, 2019.

6 S.-W. Cheng, L. Mencel, and A. Vigneron. A Faster Algorithm for Computing Straight Skeletons. *ACM Transactions on Algorithms*, 12(3):44:1–44:21, 2016.

7 G. Eder and M. Held. Computing Positively Weighted Straight Skeletons of Simple Polygons Based on Bisector Arrangement. *Information Processing Letters*, 132:28–32, 2018.

8 G. Eder, M. Held, S. Jasonarson, P. Mayer, and P. Palfrader. On Generating Polygons: Introducing the Salzburg Database. In *Proceedings of the 36th European Workshop on Computational Geometry*, pages 75:1–7, 2020.

9 D. Eppstein and J. Erickson. Raising Roofs, Crashing Cycles, and Playing Pool: Applications of a Data Structure for Finding Pairwise Interactions. *Discrete & Computational Geometry*, 22(4):569–592, 1999.

10 M. Held and P. Palfrader. Straight Skeletons with Additive and Multiplicative Weights and Their Application to the Algorithmic Generation of Roofs and Terrains. *Computer-Aided Design*, 92(1):33–41, 2017.

11 M. Held and P. Palfrader. Skeletal Structures for Modeling Generalized Chamfers and Fillets in the Presence of Complex Miters. *Computer-Aided Design and Applications*, 16(4):620–627, 2019.

12 S. Huber. *Computing Straight Skeletons and Motorcycle Graphs: Theory and Practice*. Shaker Verlag, 2012. ISBN 978-3-8440-0938-5.

13 S. Huber and M. Held. Theoretical and Practical Results on Straight Skeletons of Planar Straight-line Graphs. In *Proceedings of the 27th Symposium on Computational Geometry (SoCG)*, 2011.

14 T. Kelly and P. Wonka. Interactive Architectural Modeling with Procedural Extrusions. *ACM Transactions on Graphics*, 30(2):14:1–14:15, Apr. 2011.

15 P. Palfrader, M. Held, and S. Huber. On Computing Straight Skeletons by Means of Kinetic Triangulations. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA)*, pages 766–777, 2012.

16 The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 5.0 edition, 2019.

17 A. Vigneron and L. Yan. A Faster Algorithm for Computing Motorcycle Graphs. *Discrete & Computational Geometry*, 52(3):492–514, 2014.