FIST: Fast Industrial-Strength Triangulation of Polygons*

Martin Held†
Institut für Computerwissenschaften
Universität Salzburg
A-5020 Salzburg
Austria

Abstract

We discuss a triangulation algorithm that is based on repeatedly clipping ears of a polygon. The main focus of our work was on designing and engineering an algorithm that is (1) completely reliable, (2) easy to implement, and (3) fast in practice. The algorithm was implemented in ANSI C, based on floating-point arithmetic. Due to a series of heuristics that get applied as a back-up for the standard ear-clipping process if the code detects deficiencies in the input polygon, our triangulation code can handle any type of polygonal input data, be it simple or not. Based on our implementation we report on different strategies (geometric hashing, bounding-volume trees) for speeding up the ear-clipping process in practice. The code has been tuned accordingly, and cpu-time statistics document that it tends to be faster than other popular triangulation codes. All engineering details that ensure the reliability and efficiency of the triangulation code are described in full detail. We also report experimental data on how different strategies for avoiding sliver triangles effect the cpu-time consumption of the algorithm. Our code, named FIST as an acronym for fast industrial-strength triangulation, forms the core of a package for triangulating the faces of 3D polyhedra, and it has been successfully incorporated into several industrial graphics packages, including an implementation for Java 3D by Sun Microsystems.

Key words: Polygon, triangulation, ear clipping, geometric hashing, reliability, robustness, experimental analysis.

*A preliminary version of this paper is to appear as an extended abstract at CGI'98.

†Email: held@cosy.sbg.ac.at. Most of this work was carried out while working at the Computational Geometry Lab of SUNY Stony Brook, and was supported by grants from Sun Microsystems, and by NSF Grant CCR-9504192.

1 Introduction

1.1 Motivation

The triangulation of a polygon is a basic building block for many graphics applications. For instance, high-speed rendering typically relies on polygonal and curved surfaces being subdivided into triangles that can be handled efficiently by the graphics hardware. Triangulating a polygon also is a fundamental operation in computational geometry, and it has received wide-spread interest over the last two decades, see Bern and Eppstein's recent survey [11].

Through a series of improvements, the worst-case complexity of triangulating a polygon with n vertices has been brought down from $O(n^2)$ to the optimal $O(n)$, achieved by a seminal algorithm designed by Chazelle [16]. Algorithms with a slightly super-linear expected complexity also are known, cf. Seidel's [40] randomized algorithm.

Virtually all published triangulation algorithms assume that the polygon is simple, i.e., that the vertices of the polygon are the only points of the plane that belong to two edges, and that no point of the plane belongs to more than two edges. Due to the Jordan curve theorem, a simple polygon divides the plane into a well-defined interior and exterior. Obviously, the requirement to be simple prevents a polygon from having self-intersections. However, it also excludes so-called "degenerate" situations, such as an edge passing through another vertex, zero-length edges, and edges that partially overlap.

While avoiding the hassles involved with handling "degenerate" or "illegal" cases, the restriction to truly simple polygons constitutes a major road-block for the practical application of triangulation algorithms. Unfortunately, real-world polygons cannot be assumed to be truly simple polygons that are in general position. Rather, real-world polygons tend to exhibit all types of deficiencies, such as self-intersections, or island contours that overlap with each other or with the outer boundary.

We note that exact arithmetic does not really help with the practical handling of polygons that are not simple. For a triangulation code that tessellates the faces of a polyhedron and sends them to the subsequent rendering hardware it does not matter whether a polygon is classified as nearly degenerate but still simple, or as slightly self-intersecting — in any case, the code must not crash or loop, and a reasonable triangulation has to be computed. And, of course, this triangulation has to be computed without human intervention.

This paper presents an efficient triangulation algorithm that is able to cope with any form of polygonal input data, be it simple or not. Specifically,

- we described a *triangulation algorithm based on ear clipping*;
- we give a *detailed discussion of reliability issues* related to the handling of "degenerate" or "illegal" polygonal input, and explain how a robust and reliable code has been engineered; and
- we provide a *thorough experimental analysis* of how to use geometric hashing in order to make the algorithm competitive with algorithms that exhibit a better worst-case behavior.

1.2 Survey of the Algorithm

In the sequel, a polygon is denoted by $\mathcal{P}$, and n stands for the number of its vertices. As a convention, we include the interior of a polygon when referring to $\mathcal{P}$. Unless stated otherwise, we will assume that $\mathcal{P}$ is a simply-connected area, i.e., bounded by one single loop of straight-line segments. The vertices are numbered in counter-clockwise orientation and denoted by $v_0, \ldots, v_{n-1}$. As usual, all vertex indices are taken modulo n.

We present an algorithm that triangulates a polygon by searching for "ears" and "clipping" them. As usual, three consecutive vertices v_{i-1}, v_i, v_{i+1} of $\mathcal{P}$ are said to form an *ear* of $\mathcal{P}$ if v_{i-1}, v_{i+1} is a diagonal of $\mathcal{P}$. According to Meisters [33], every simple polygon that is not a triangle has at least two non-overlapping ears.

A straightforward implementation of a triangulation algorithm based on repeatedly clipping ears runs in $O(n^3)$ time, with $O(n)$ time spent on checking whether three subsequent vertices form an ear. A simple re-organization of the ear-finding process allows to check only $O(n)$ ears, thus finding all ears in $O(n^2)$ time, see [31, 36]. However, this complexity still is too high for practical applications. Whereas faces of a polyhedron typically have rather few vertices, applications in GIS may involve polygons with several tens of thousands of vertices.

We try to reduce the practical complexity of finding one ear by performing only local searches. Bounding-volume trees, as successfully used in our work on collision detection [30, 26], or, alternatively, regular grids are used for registering the polygon under triangulation and for pruning the search while testing for earity[1]. While any form of geometric hashing does not reduce the worst-case complexity, it helps a lot to reduce the practical running time of the triangulation algorithm, as witnessed by our experiments.

The algorithm first determines the orientation of the polygon. Then it repeatedly clips ears of the polygon until the polygon has been transformed to a triangle and the entire triangulation is known. The actual core of the algorithm is based on one predicate: we only need to check the sidedness of three points. (In addition, a lexicographical sort of the vertices of the polygon is used.) We make sure that this primitive is always used consistently. Furthermore, care has been taken to ensure that degenerate cases are handled correctly.

If the algorithm cannot find any valid ear to clip — e.g., due to deficiencies of the input polygon, or due to numerical errors — then it tries to restart the triangulation process by applying several heuristics. For instance, it checks whether subsequent edges of the polygon intersect, and applies a special rule in order to by-pass this deficiency. Then, the standard ear-clipping process is resumed again. Similarly, it tries to find a diagonal that splits the polygon into two sub-polygons, which are then again subjected to the standard ear-clipping process.

If everything fails, and no heuristic helps to reduce the number of vertices of the polygon, then the algorithm (temporarily) goes into "desperate mode". It randomly picks a convex vertex, and clips the corresponding "ear". If no convex vertex exists then a reflex vertex is chosen randomly. Thus, the more desperate the code gets the more aggressive means are used for "finding" an ear or a diagonal, thereby always reducing the number of vertices of $\mathcal{P}$ by at least one. This strategy allows the code to produce triangulations that degrade gracefully as the deficiencies of the input polygon get more serious. Of course, at

[1] "Earity test" is the common nickname for the process that tests whether a triple of consecutive vertices forms an ear.

some point the old principle of "garbage in, garbage out" applies. However, the code will always output a triangulation that is topologically valid.

The algorithm has been implemented in C, and extended to handle multiply-connected polygons. The resulting software, called FIST as acronym for "Fast Industrial-Strength Triangulation", has been tested extensively, and we present statistics on a series of test runs. Using synthetic test data generated by means of RPG [1, 2], we compared the cpu-time consumption of FIST and to those of other popular C codes: we compared it to Narkhede and Manocha's implementation [34, 35] of Seidel's algorithm [40], Shewchuk's "Triangle" [43, 41], Sloan's implementation [45] of an ear-clipping algorithm, and to Saade's implementation [39] of Toussaint's algorithm [47]. As witnessed by our tests, FIST is competitive with the other codes, and even outperformed them in most tests.

This paper is accompanied by several color plates available on the WWW. Point your browser to the WWW home-page [27] of FIST.

The remainder of this paper is organized as follows. We discuss prior and related work in the following subsection. Section 2 contains a presentation of the basic triangulation algorithm. The next two sections, Section 3 and Section 4 discuss enhancements of the basic algorithm for improving its speed and its reliability. Experimental results are given in Section 5, and Section 6 reports on tests and experiments for determining the practical reliability of the code. The paper is concluded in Section 7.

1.3 Prior and Related Work

Most early triangulation algorithms try to find a diagonal of the polygon and then recurse on the two resulting sub-polygons. Meisters [33] proved that every simple polygon that is not a triangle has at least two non-overlapping ears. Since finding an ear takes $O(n)$ time in the worst case, ear-clipping algorithms need at least $O(n^2)$ time in the worst case. Several $O(n^2)$ algorithms based on ear clipping have been proposed recently, see [31, 22, 36].

Garey et al. [25] were the first to publish an $O(n \log n)$ triangulation algorithm. They used a regularization scheme in order to partition a polygon in $O(n \log n)$ time into monotone sub-polygons. Monotone sub-polygons are afterwards triangulated in linear time by repeatedly clipping off convex corners (i.e., ears) of the polygon. See also the work by Fournier and Montuno [24]. A completely different algorithm by Chazelle [15], based on divide-and-conquer, also achieves an $O(n \log n)$ worst-case bound.

The first triangulation algorithm that is sensitive to the shape of the polygon was given by Hertel and Mehlhorn [28]. Their algorithm achieves a complexity of $O(n+r \log r)$, where r denotes the number of reflex vertices of the polygon. See also the simple algorithm by Kong et al. [31] which runs in $O(n \cdot (r+1))$ time.

The algorithm by Chazelle and Incerpi [17] also takes advantage of the shape of the polygon. Their algorithm, which is based on trapezoidal decompositions, runs in time $O(n \log s)$, where s is the sinuosity of the polygon. (Roughly, the sinuosity of a polygon gives the number of times the polygon's boundary alternates between spirals of opposite direction.)

Toussaint's algorithm [47] is another algorithm that is shape-dependent. It runs in time $O(n + nt_0)$, where t_0 is the number of triangles of the triangulation that have three neighbors (i.e., that share no edge with the polygon).

Seidel [40] described a simple randomized incremental algorithm for triangulating polygons. Its expected complexity is $O(n \log^\star n)$. Thus, its expected complexity is almost linear for any simple polygon. This algorithm first incrementally computes a trapezoidal decomposition of the polygon, by taking a random permutation of the edges of the polygon and inserting one segment at a time into this decomposition. The number of trapezoids is linear in n, and Seidel proved that computing this trapezoidal decomposition takes $O(n \log^{\star} n)$ expected time if each permutation of the edges of the polygon is equally likely. Based on the trapezoidal decomposition a set of monotone polygonal chains is obtained in linear time. Finally, these chains are triangulated in linear time by using the algorithm by Fournier and Montuno [24].

An earlier randomized triangulation algorithm, also with $O(n \log^{\star} n)$ expected complexity, was published by Clarkson et al. [19]. See also the work by Devillers [20].

Tarjan and Van Wyk [46] were the first to break the $O(n \log n)$ worst-case bound. Their algorithm runs in $O(n \log \log n)$ time. A simpler algorithm that achieves the same worst-case time bound was later published by Kirkpatrick et al. [29].

Finally, Chazelle [16] ended the quest for a worst-case optimal triangulation algorithm. His ingenious construction, based on coarse approximations of visibility maps, runs in $O(n)$ time. However, it seems to be too complicated to be implemented successfully.

The situation is quite different when dealing with polygons in 3D. Call a 3D polygon triangulable if it has a triangulation which is not self-intersecting and which defines a simply-connected 2-manifold. Then, Barequet et al. [6] show that the problem of deciding whether a 3D polygon is triangulable is $\mathcal{NP}$-complete. Also, they give an $O(n^4)$ algorithm for determining a plane on which a projection of a given polygon is simple.

Related to the triangulation of a 3D polygon is the linear interpolation of polygonal slices in 3D. Barequet and Sharir [8] use dynamic programming for determining a minimum-area triangulation between two polygonal slices. As is well known, dynamic programming requires $O(n^3)$ time and $O(n^2)$ space. See also the work by Barequet et al. [9, 10, 7], and Choi and Park [18].

A planar multiply-connected polygon with i islands and a total of n edges can be triangulated in time $O(n \log n)$ by applying any of the standard techniques, such as plane sweep or regularization. The best bound known for the worst-case complexity is achieved by an algorithm due to Bar-Yehuda and Chazelle's [5], which runs in (near-optimal) time $O(n + i \log^{1+\epsilon} i)$.

Ronfard and Rossignac [38] describe a finite-state machine that is able to handle polygons with islands. Their algorithm, called "flooding", is similar in spirit to sweep-line algorithms, but it sweeps only subparts ("gorges") of the polygon at a time. This algorithm was implemented at IBM, and the authors report an average complexity of $O(n^{3/2})$, with a worst-case complexity of $O(n^2)$.

There seem to be far more theoretical studies of triangulation algorithms than published reports on actual implementations, anyway. Seidel's algorithm [40] got implemented by Narkhede and Manocha [34], and their code is available publicly [35]. (Since its first publication, this code has been extended to handle polygons with islands, and has been tuned in order to reduce its start-up cost.) An implementation of Toussaint's algorithm [47] was carried out by Saade [39].

Sloan [45] implemented an $O(n^2)$ triangulation algorithm based on ear clipping. His algorithm also is able to swap diagonals in order to avoid skinny triangles, and can group triangles into quadrangles.

Shewchuk [43] implemented "Triangle", a fine code that computes Delaunay triangulations and is able to perform guaranteed-quality meshing. "Triangle" is based on geometric predicates that rely on an adaptive floating-point arithmetic, cf. [42]. This code is also available publicly [41].

The issues of exact computation versus floating-point arithmetic are discussed in detail in a survey by Yap and Dubé [49]. Recent work includes papers by Avnaim et al. [3] and Brönnimann and Yvinec [13] on the exact evaluation of integer determinants, Brönnimann et al. [12] on exact geometric predicates with single-precision arithmetic, and Shewchuk's design and implementation [42] of four predicates based on adaptive floating-point arithmetic. See the recent survey papers by Yap and Dubé [49, 48]. Exact arithmetic is offered by the geometric software packages LEDA [14, 32] and CGAL [23, 37].

Recently, Silva et al. [44] have used an approach similar in spirit to the one described in this paper for ensuring reliability. While not addressing the robustness issue directly, they also describe how a carefully tailored ear-clipping algorithm, combined with some additional heuristics, has yielded a fairly reliable and practical algorithm for generating triangular irregular networks (TINs).

2 Triangulation Algorithm

2.1 Basic Algorithm

This subsection describes the basic ear-clipping algorithm. We will first assume that $\mathcal{P}$ is a truly simple polygon which does not exhibit any degeneracies. Also, $\mathcal{P}$ shall be oriented counter-clockwise.

As usual, a vertex v of $\mathcal{P}$ is called *convex* if the internal angle of $\mathcal{P}$ at v is less than 180^o, and *reflex* otherwise. Reflex vertices with an internal angle equal to 180^o are also called *tangential*. Two vertices $v_i \neq v_k$ form a *diagonal* of $\mathcal{P}$ if the relative interior of the line segment $[v_i, v_k]$ is completely contained in the interior of $\mathcal{P}$. For a convex vertex v_i, the *cone* defined by the three consecutive vertices v_{i-1}, v_i, v_{i+1}, denoted by $C(v_{i-1}, v_i, v_{i+1})$, is the set of points that are strictly right of the oriented line v_i, v_{i-1} and strictly left of the oriented line v_i, v_{i+1}. Similarly, if v_i is reflex then the cone defined by v_{i-1}, v_i, v_{i+1} is the complement of the union of the cone defined by v_{i+1}, v_i, v_{i-1} and the two rays from v_i through v_{i-1} and v_{i+1}.

Recall that three consecutive vertices v_{i-1}, v_i, v_{i+1} form an *ear* of $\mathcal{P}$ if v_{i-1}, v_{i+1} is a diagonal of $\mathcal{P}$. Obviously, for v_{i-1}, v_i, v_{i+1} to form an ear, v_i has to be a convex vertex. Also, $[v_{i-1}, v_{i+1}]$ can only be a diagonal if it is locally inside $\mathcal{P}$. That is, v_{i+1} has to lie in $C(v_{i-2}, v_{i-1}, v_i)$, and v_{i-1} has to lie in $C(v_i, v_{i+1}, v_{i+2})$. Finally, v_{i-1}, v_{i+1} must not intersect the boundary of $\mathcal{P}$ except in the edges (v_{i-1}, v_i) and (v_i, v_{i+1}), or, equivalently, the triangle $\Delta(v_{i-1}, v_i, v_{i+1})$ must not contain any reflex vertex of $\mathcal{P}$. (Note that these two conditions are only equivalent for simple polygons.)

Thus, we have the following two sets of conditions for v_{i-1}, v_i, v_{i+1} to form an ear [31, 47, 36]. Both sets of conditions are necessary and sufficient.

Fact 2.1 (CE1) Three consecutive vertices v_{i-1}, v_i, v_{i+1} of $\mathcal{P}$ form an ear of $\mathcal{P}$ iff

1. v_i is a convex vertex;

2. $v_{i-1} \in C(v_i, v_{i+1}, v_{i+2})$ and $v_{i+1} \in C(v_{i-2}, v_{i-1}, v_i)$;

3. the segment $[v_{i-1}, v_{i+1}]$ does not intersect any edge of $\mathcal{P}$ except in v_{i-1} or in v_{i+1}.

Fact 2.2 (CE2) Three consecutive vertices v_{i-1}, v_i, v_{i+1} of $\mathcal{P}$ form an ear of $\mathcal{P}$ iff

1. v_i is a convex vertex;
2. $v_{i-1} \in C(v_i, v_{i+1}, v_{i+2})$ and $v_{i+1} \in C(v_{i-2}, v_{i-1}, v_i)$;
3. the closure of the triangle $\Delta(v_{i-1}, v_i, v_{i+1})$ does not contain any reflex vertex of $\mathcal{P}$.

We have implemented ear clipping according to both CE1 and CE2. Both have a straightforward implementation that checks Condition (3) by a brute-force scan of all segments, respectively of all reflex vertices. For a k-gon with r reflex vertices, the time complexity of this check is $O(k)$ for CE1, and $O(r)$ for CE2. In Subsection 3.1, we will discuss heuristics for decreasing the practical complexity of checking Condition (3).

The actual triangulation algorithm works as follows: After classifying the vertices of $\mathcal{P}$ as convex or reflex, we check for every triple of consecutive vertices of $\mathcal{P}$ whether they form an ear. Ears are marked and stored. (In Subsection 3.2 we will discuss the storage and retrieval of ears in more detail.) Then the code iteratively retrieves one of the stored ears, clips it, and updates the information on ears. It is well known that clipping the ear v_{i-1}, v_i, v_{i+1} transforms a k-gon into a $k-1$-gon, for which all previously determined ears still are valid, except for up to two ears that involve v_i. (Since v_i got deleted, v_{i-2}, v_{i-1}, v_i and v_i, v_{i+1}, v_{i+2} cannot be any ears of the new polygon.) In addition, $v_{i-2}, v_{i-1}, v_{i+1}$ and $v_{i-1}, v_{i+1}, v_{i+2}$ could now form ears and have to be checked. The algorithm iteratively cuts ears until the original n-gon has been transformed into a 3-gon (i.e., a triangle), and the last triangle of the triangulation is trivially known.

The correctness of the algorithm is established similar to other published ear-clipping algorithms, cf. [31, 36]. As analyzed in [31, 47, 36], the worst-case complexity of this algorithm is $O(n^2)$ if CE1 is used for determining ears, and $O(n \cdot (r+1))$ if CE2 is used.

2.2 Ensuring Consistency

Before the ear-clipping process starts, we sort the vertices of the polygon lexicographically, first according to x-coordinates, and second according to y-coordinates. A simple scan through the array of sorted vertices reveals all true duplicates[2], which are discarded in the array. Then, by means of a binary search, every vertex of the polygon gets assigned its index in the sorted array of vertices. As a result, vertices that have identical coordinates have the same index. (If desired by the application, we can now also delete any edge whose start and end vertices are identical, i.e., zero-length edges.) In the sequel, whenever we speak of the index of a vertex we will refer to its index in this sorted array.

A close inspection of the computations required by the ear-clipping algorithm reveals that one basic primitive operation suffices: all we need to do is to check whether some vertex v_i is left of, on, or right of the oriented line through two other vertices v_j and v_k. As usual, this primitive is implemented as the evaluation of the sign of a 3×3 determinant.

Care is taken that the evaluations are consistent. We always compute determinants such that the index of the first vertex is less than the index of the second vertex, which in turn is less than the index of the third vertex. If this causes a change of the cyclic order of the vertices then the sign of the resulting determinant is inverted. (Trivially, the determinant is zero if any two indices of the vertices are identical.)

[2] I.e., vertices which have identical coordinates.

The actual sign of the determinant is computed by comparing it to a user-specified threshold value ("epsilon"). Numerical errors in the evaluation of the determinant may result in an incorrect judgment of whether the sign is -1,0, or +1. However, it will be consistent throughout the entire algorithm. Also, note that the choice of a suitable epsilon is not critical for the code. It will work with any non-negative epsilon. Of course, the actual triangulation computed will depend on the specific value of this epsilon, as it will influence which points are considered to be collinear.

It is tempting to try to prove that an ear-clipping algorithm that relies on such a primitive will manage to triangulate any simple polygon, where simplicity is defined relative to this primitive. That is, if the primitive classifies no two (non-consecutive) edges of the polygon as intersecting then one should also be able to determine a triangulation based on ear clipping. Unfortunately, this is not true. If the primitive is allowed to err, then it is easy to see that we can assign consistent signs to our primitive such that the edges of a hexagon will not intersect pair-wise except at their mutual end points, but such that the hexagon will not have a single ear according to CE1 or CE2.

2.3 Handling Multiply-Connected Polygons

The basic ear-clipping algorithm can be extended to multiply-connected polygonal areas. It was not the main purpose of this work to design and implement a fast algorithm for polygons with lots of islands, though. Our goal was to extend the ear-clipping algorithm to multiply-connected areas in a reliable way with minimal cpu-time consumption, without resorting to any deep theoretical results on how to triangulate multiply-connected areas.

Thus, we have contented ourselves with transforming a multiply-connected polygon into a simply-connected polygon by iteratively linking the island loops with the outer boundary loop by means of contour "bridges". A "bridge" is a pair of parallel segments, and inserting a bridge transforms a multiply-connected polygon with one island into a simply-connected polygon. Formally speaking, the resulting polygon is not truly simple as part of its boundary overlaps (at the bridges). After bridges have been inserted the standard ear-clipping algorithm can be applied.

Finding a bridge means checking for nodes v and v' (that are on different polygonal loops) whether the segment $[v, v']$ does not intersect any loop except at v and v'. A naive implementation of bridge finding would thus consume $O(n^3)$ time. (There are $O(n^2)$ many potential candidates, and it takes linear time to check the validity of one bridge candidate.)

Our approach tries to find one bridge in roughly $O(n \log n)$ time. For every island loop we determine its left-most vertex. This is readily (and consistently) accomplished by comparing the indices of the vertices. Then we sort the islands according to their left-most vertices. The left-most island is the first to be linked with the outer boundary. Let v be the left-most vertex of this island. All vertices of the outer boundary that are left of v are sorted according to increasing distance from v. (In order to save cpu time, the actual implementation uses the L_1-norm instead of the Euclidean norm.) Recall that a vertex v' is left of v if its index is smaller than the index of v.

Starting with the closest vertex, v', we test whether $[v, v']$ forms a bridge (diagonal) between the outer boundary and the island loop. The motivation for sorting the vertices according to their distance from v is that a near-by vertex is likely to be a feasible vertex for a bridge to v. Thus, hopefully, only few pairs v, v' need to be tested for feasibility. Also, note that the segment $[v, v']$ cannot intersect the current island loop (or any other island loop) if v' is left of v.

Obviously, the worst-case complexity of this simple scheme is $O(i \cdot n^2)$ for a multiply-connected polygonal area with i islands. (Checking one pair v, v' may need $O(n)$ time, and we may have to check up to n pairs per island.) However, this simple approach works quite well for areas that have only a few islands. As explained in Subsection 3.1, geometric hashing helps to speed up things in practice. Also, our approach to finding bridges is in line with the consistency checks of the rest of the code.

3 Improving the Triangulation Algorithm

3.1 Geometric Hashing for Faster Ear Clipping

Next to achieving a reliable algorithm, it is important to get rid of the $O(n^2)$ term that renders the standard ear-clipping approach infeasible for anything but small polygons. (See Section 5.3.) Depending on whether ears are determined according to CE1 or CE2, we have investigated different means for reducing the number of operations needed for an earity test: we studied bounding-volume trees (BV-trees) and regular grids. We describe their principles below; run-time statistics are reported in Section 5.

CE1 and BV-trees: We use the same hierarchical bounding-volume technique that we have successfully applied to collision detection in 3D, cf. [30, 26]. The root of the BV-tree contains the (axis-aligned) bounding box of the entire polygon. Then we split the edges of the polygon into two groups – a "left" group and a "right" group, or a "top" and a "bottom" group – by comparing the x-coordinate (or the y-coordinate) of the mid-point of every edge to the mean x-coordinate (resp., y-coordinate), where the mean is taken over the midpoints of all edges. We use the split which minimizes the sum of the areas of the resulting two bounding boxes, and recurse until every bounding box contains only one edge of the polygon. Thus, the bounding boxes of the individual edges form the leaves of our BV-tree.

Checking whether a query segment (defined by two vertices of the polygon) intersects an edge of the polygon is done by a depth-first search of the BV-tree. The bounding box of the query segments is pushed down the BV-tree until no overlap with any node of the tree exists, or until the first leaf is reached. Then, a conventional intersection check is carried out between the two line segments. If no intersection is found then the next leaf whose bounding box overlaps the bounding box of the query segment is determined, and another intersection check is carried out. This goes on until an intersection has been found or until all edges whose bounding boxes overlap the bounding box of the query edge have been checked.

Since the polygon changes after every ear-clipping step we have to think about handling insertions of edges into and deletions from the BV-tree. We have implemented one version that handles those updates of the BV-tree. Alternatively, we may choose to ignore the fact that the polygon has changed after an ear-clipping operation. It is easy to see that we can also check any new potential diagonal for intersection with the original polygon[3], rather than with the actual one. Thus, we have also implemented a version that does not perform insertions and deletions after one ear has been clipped, but rather computes the

[3]Whether or not the original edges are used only makes a difference for polygons that are not simple.

entire BV-tree from scratch ever so often. (After a series of experiments we decided to update the BV-tree whenever $\frac{n}{\log \log n}$ ears have been clipped.)

CE1 and Grids: Rather than using a BV-tree, we can also store the edges of the polygon in a regular rectangular grid that subdivides the bounding box of the polygon. Thus, for every edge of the polygon we store a pointer to this edge in those cells of the grid that the edge passes through. Any standard scan-conversion algorithm for rasterizing line segments with respect to a coarse grid suffices to determine the cells intersected by an edge. Since the grid boundaries are not part of our input we prefer to err on the safe side and register an edge with two neighboring cells if the edge lies "close" to the cell boundary between those two cells.

Using the grid, an intersection check between a query segment and the edges of the polygon is reduced to checking the edges of those cells that the segment passes through. Since we expect an edge of the polygon to pass through only a few cells we do not employ any "mail-box" concept in order to avoid re-checking an edge multiple times. Similar to the case of BV-trees, we experimentally learned that it does not pay off to update the grid data structure accordingly as ears of the polygon get clipped.

Predicting a suitable resolution of the grid is not exactly easy as it may depend on the distribution of the edges of the polygon. Clearly, we cannot afford to waste memory on a very fine grid. Thus, we decided that the grid should have roughly $w\sqrt{n} \times h\sqrt{n}$ cells for a polygon with n vertices, where w and h depend on the aspect ratio of the bounding box of the polygon. After a series of experiments (see Section 5) we decided to choose w and h such that $w \cdot h = 2.5$. Thus, our grid consists of roughly $2.5n$-many cells, and the additional memory consumption caused by the grid is negligible on a standard workstation.

CE2 and Grids: Clipping ears according to CE2 requires to check whether a triangle defined by three consecutive vertices of the polygon contains any reflex vertex of the polygon. Thus, for every reflex vertex of the polygon we determine the cell that contains it and store the vertex with this cell. (If a vertex lies "close" to a cell boundary then we also store it with the neighboring cell(s).)

For a CE2 query we first determine the bounding box of the triangle. Then, we check all reflex vertices that are stored with cells overlapped by this bounding box. The resolution of the grid is again given by $w\sqrt{n} \times h\sqrt{n}$. For this grid we have experimentally determined that $w \cdot h = 1$ is best, see Section 5.

An update of the grid data structure is comparatively easy and efficient to perform. For every ear clipped we have up to two reflex vertices that may have been transformed into convex vertices, and thus need to be deleted from the grid. Practical experience with our code taught us that deleting reflex vertices that do no longer belong to the polygon may help in finding valid ears in the case of self-overlapping polygons.

As documented in Section 5, using a grid for checking the conditions of CE2 turned out to be the fastest variant of the ear-clipping algorithm. Thus, in the sequel we fill mainly focus on clipping ears according to CE2.

3.2 Avoiding Sliver Triangles

The standard ear-clipping algorithm tends to generate triangulations that contain quite a few sliver triangles. In this so-called *sequential* mode, the code would clip an ear, check whether the two neighboring vertices form ears, and, if they did indeed form ears, would proceed with clipping one of those two new ears. As a result, a convex polygon ends up with a fan triangulation, with all triangles centered at one vertex. While sliver triangles may not cause much harm in some graphics applications (such as rendering), there do exist applications (such as FEM analysis) for which sliver triangles are highly undesirable.

This prompted us to implement two other heuristics for selecting the next ear to be clipped. Running the code in *random* mode causes the code to randomly pick an ear among all feasible ears. In *sorted* mode, the code uses a greedy strategy and always clips the ear which is nicest, i.e., which best resembles an equilateral triangle. Whenever a new ear is found we compute a numerical shape classifier for this ear, and the ears are arranged in a priority queue according to this numerical classifier such that the nicest ears are at the front of the queue.

Both heuristics, *random* and *sorted*, help significantly in avoiding sliver triangles, at the expense of a slightly increased cpu-time consumption. See Section 5 for statistical data on our experiments.

We emphasize that any of these heuristics for avoiding sliver triangles can be combined with any of the hashing techniques outlined in Section 3.1. Typically, the standard ear-clipping algorithm as described in the literature [31, 36] corresponds to sequential ear clipping according to CE1 or CE2, with no geometric hashing used.

4 Coping with Data Deficiencies

Real-world data tends to exhibit all types of deficiencies, and a triangulation code that is supposed to handle real-world polygons needs special care that goes well beyond a flawless implementation of the basic algorithm. Typical examples of deficiencies in polygonal data include so-called "degeneracies" — such as zero-length edges, vertices that lie on other edges of the polygon, edges that overlap, 0^o-degree and 360^o-degree internal angles — as well as more apparent violations, such as true self-intersections, overlaps between the boundary loop and island loops, incorrectly specified orientations of polygons, and polygons that are twisted (such as a figure-of-eight). In the sequel, we will classify a polygon as *degenerate* if it does not qualify as a simple polygon, but if an arbitrarily small perturbation of its vertices would transform it into a simple polygon. Clearly, polygons that are twisted or that have self-overlapping parts do not qualify as degenerate.

Whereas some deficiencies may be intentional (such as a grazing contact between a vertex and an edge), most deficiencies are the result of numerical or algorithmic problems of the software that generated the polygons. For instance, many solid modelers still seem to have troubles with generating consistent and clean polyhedral approximations of curved objects. In any case, be the deficiency intentional or by mistake, our triangulation algorithm has to cope with it.

We note that handling polygonal deficiencies cannot be accomplished by resorting to using exact arithmetic instead of the conventional floating-point arithmetic. Vertices may truly lie on other edges, and interior angles may indeed be 0^o or 360^o. Also, the use of symbolic perturbation[4] does not solve the problem as it may transfer a barely legal polygonal structure into something that clearly is no simple polygon any more, thus aggravating the

[4]For instance, see Edelsbrunner and Mücke [21].

situation. The only remedy is to design an algorithm that is able to cope with any form of polygonal input.

4.1 Orientation of the Polygonal Loops

After the vertices have been sorted lexicographically, our algorithm first determines the orientation of every polygonal loop. Several approaches to determining the orientation of a polygon have been proposed, e.g., see the recent paper by Balbes and Siegel [4]. We settled on computing the signed area of a polygon for determining its orientation. The area of a polygon is computed by summing over the signed areas of the triangles $\Delta(v_0, v_i, v_{i+1})$. Obviously, the area of a simple polygon is positive if it is oriented counter-clockwise (CCW), and negative for clockwise (CW) polygons. Computing the area of a polygon is not effected by degeneracies, and it also allows to make an educated guess of what should be the orientation of a polygon that is twisted: For a polygon that forms a figure-of-eight, that part of the polygon that encloses the larger area determines the orientation of the polygon.

If the input consists of more than one polygonal loop then we also need to determine which loop forms the outer boundary loop, i.e., which polygon contains the other polygons. We compare the (absolute values of the) computed areas of the polygon, the rationale being that the polygon with maximum area has to form the outer boundary. This conclusion is obviously true for multiply-connected areas without deficiencies, and it provides a reasonable guess even if the island loops overlap with each other or with the outer boundary.

As soon as the outer boundary loop is determined, we orient all loops such that the outer loop is oriented CCW, and such such all island loops are oriented clockwise. If the area of a polygon happens to be zero then a random orientation is chosen. (Then, either the polygon is collapsed to a polygonal chain, or it is twisted. In either case, it is likely not to matter which orientation is chosen.)

As with determining the orientation of a polygon we originally tried to save cpu time by using only local evaluations that do not involve the entire polygon. However, we quickly learned that local tests cannot cope with data deficiencies and would yield judgments that (intuitively) were clearly incorrect for polygonal curves that were slightly messed up. We emphasize, though, that our approach to determining the orientations and the containment relation of polygonal loops is not guaranteed to make the "correct" judgment call for any form of twisted or self-overlapping polygonal input data. It works very well for minor twist or overlaps, though, and seems to classify them correctly according to the user's intuition. And, in particular, it yields the correct result for polygonal input data which has no deficiencies.

4.2 Classification of Internal Angles

The actual ear-clipping process also needs a few amendments. First of all, while classifying vertices as convex or reflex (by means of our determinant primitive) we have to handle the case when this determinant is (nearly) zero. This indicates that the internal angle is 0^o, 180^o, or 360^o. Obviously, the internal angle at v_i is 0^o if the vectors from v_i to v_{i-1} and to v_{i+1} point into opposite directions, which can be judged reliably by computing the dot product of those vectors.

If those vectors point into the same direction then the angle is 0^o or 360^o. Unfortunately, it cannot be judged locally whether the angle is 0^o or 360^o. Rather, we have to move away from v_i in CCW and in CW direction until we encounter two edges of the polygon that do not overlap (except in one end point). At this point, via a lengthy case analysis, the angle at v_i can be classified.

4.3 Ear Clipping and Degeneracies

Trying to cope with degenerate polygonal input quickly turns out to be a challenging task. In particular, the test whether three consecutive vertices form an ear needs to be adapted appropriately. Obviously, we will want to accept v_{i-1}, v_i, v_{i+1} as an ear if $v_{i-1} = v_i$ or if $v_i = v_{i+1}$, where equality among vertices is checked by comparing their indices. Similarly, v_{i-1}, v_i, v_{i+1} can still form a valid ear even if v_{i-1} lies on the edge (v_{i+1}, v_{i+2}), thus violating the in-cone condition of CE1 or CE2. In particular, v_{i-1} may equal v_{i+1} or v_{i+2}. (A symmetric condition holds for v_{i+1} and the edge (v_{i-2}, v_{i-1}).) In any of these cases we ignore the *random* or *sorted* selection schemes for picking the next ear and rather clip such a degenerate ear on the spot. This preferential treatment of degenerate ears not only helps the code to get through some degeneracies quickly, but it also is important for handling more complicated degeneracies, as explained below.

We also have to be careful when checking whether a reflex vertex v_k lies in the interior or on the boundary of a potential ear v_{i-1}, v_i, v_{i+1}, cf. Fig. 1. We do not want to accept $[v_{i-1}, v_{i+1}]$ as a diagonal if v_k lies in its relative interior (Fig. 1(a)). Also, it is easy to see that v_{i-1}, v_i, v_{i+1} does not form an ear if v_k lies in the relative interior of the edge (v_{i-1}, v_i) or in the relative interior of the edge (v_i, v_{i+1}).

The situation gets more complicated when v_i and v_k coincide. Whereas v_{i-1}, v_i, v_{i+1} constitutes an ear in Fig. 1(d), we do not want to accept this vertex triple as an ear in the situation depicted in Fig. 1(c). A test whether the line segment $[v_{i-1}, v_{i+1}]$ intersects either of the edges (v_{k-1}, v_k) or (v_k, v_{k+1}) may already help to rule out $[v_{i-1}, v_{i+1}]$ as a valid diagonal. However, v_{k-1} may be collinear with v_i, v_{i+1}, and v_{k+1} may also be collinear with v_i, v_{i-1}. In this case, a decision based on an examination of the local geometry is not possible. We again resort to computing signed areas: we determine the signed area A_{ik} of the polygon $(v_i, v_{i+1}, \ldots, v_{k-1}, v_k)$, and the signed area A_{ki} of the polygon $(v_k, v_{k+1}, \ldots, v_{i-1}, v_i)$. Only if both A_{ik} and A_{ki} are positive then we do not accept v_{i-1}, v_i, v_{i+1} as an ear. This corresponds to the situation depicted in Fig. 1(c).

Of course, one of the two areas could be zero. For instance, $(v_k, v_{k+1}, \ldots, v_{i-1}, v_i)$ could collapse to a zero-area polygonal chain. In this case, we still would want to classify v_{i-1}, v_i, v_{i+1} as an ear for the situation depicted in Fig. 1(d). Note that our rule would also incorrectly classify v_{i-1}, v_i, v_{i+1} as an ear for the situation depicted in Fig. 1(c), if A_{ki} were zero. However, the area of a polygon that is not twisted can only be zero if the polygon collapses to a polygonal chain, in which case there has to exist at least one vertex with a zero-degree internal angle. Thus, due to our rules for giving preferential treatment to degenerate ears, the code would consecutively clip all ears of the sub-polygon $(v_k, v_{k+1}, \ldots, v_{i-1}, v_i)$ before it would consider the "ear" v_{i-1}, v_i, v_{i+1}. Obviously, at that point v_{i-1}, v_i, v_{i+1} does not belong to the polygon any more, and no harm has been done by its incorrect classification.

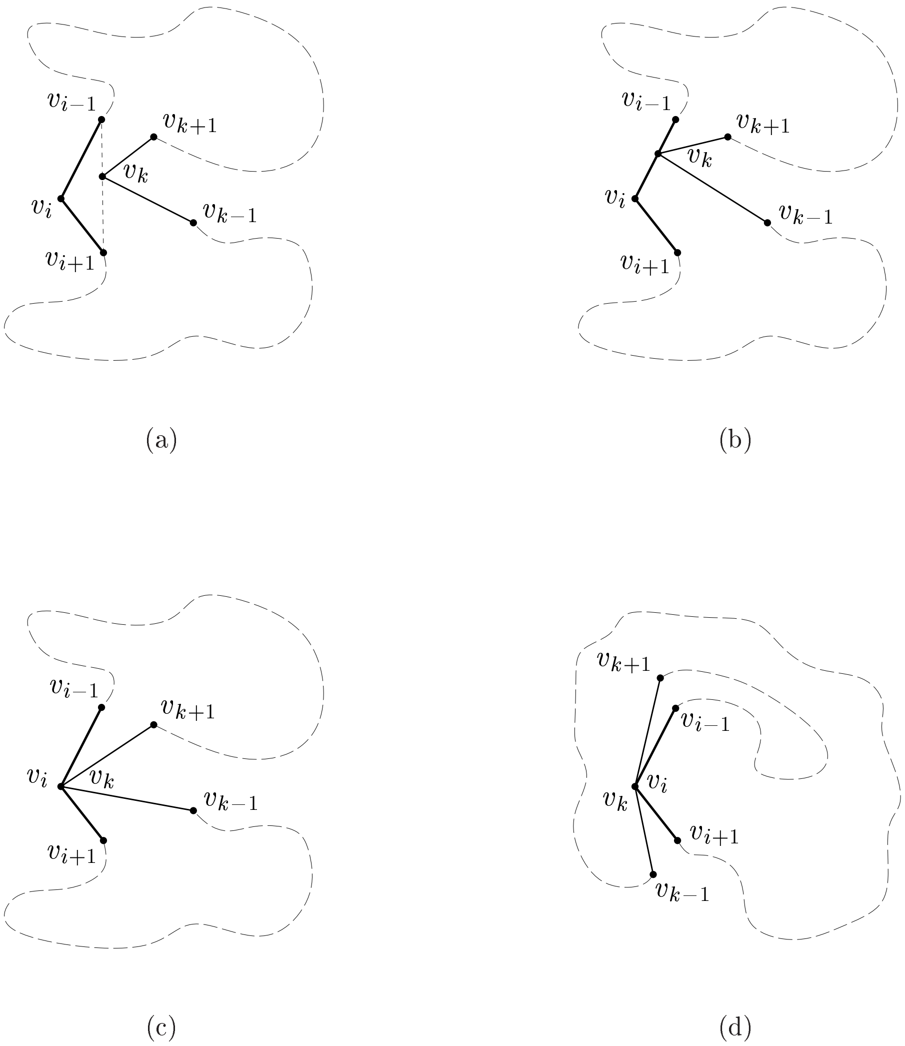


Figure 1: Ear-clipping for a degenerate polygon.

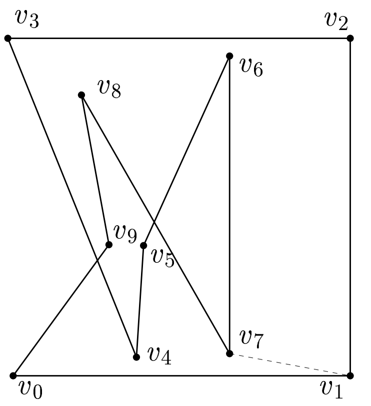
4.4 Desperate Mode

It does not come as a surprise that the ear-clipping process may run out of valid ears when applied to a polygon that is self-overlapping or that is twisted. (There exist self-overlapping polygons, though, that can be triangulated by the standard ear-clipping process.) If the ear-clipping process runs out of valid ears before the entire polygon has been triangulated, then we start a multi-phase recovery process that selects the more aggressive means for finding ears the more obvious it gets that the polygon is messed up.

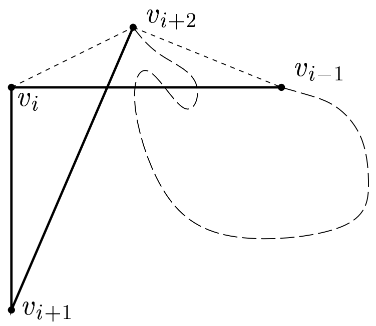
We first re-classify all ears. This is motivated by the fact that some reflex vertices of a self-intersecting polygon might already have disappeared, thus potentially rendering some vertex triples feasible as ears. If this re-classification fails to produce new ears then we try to split the polygon into two parts by inserting a diagonal. We emphasize that care has to be taken in order to insure that the insertion of this diagonal does not make things worse. In particular, a line segment $[v_i, v_j]$ need not be a valid diagonal for a self-overlapping polygon even if it is locally within the polygon, and does not intersect any edge of the polygon. For instance, consider Fig. 2(a), which depicts a polygon that has no ears according to CE2. The insertion of the diagonal $[v_1, v_7]$ splits the polygon into two sub-polygons that can then easily be handled by means of the standard ear-clipping process. However, the insertion of the line segment $[v_5, v_9]$, which also locally lies within the polygon, would split the polygon into two twisted sub-polygons. In order to avoid twisted sub-polygons we check

1. whether the potential diagonal locally is within the polygon,
2. whether its relative interior does not intersect any edge of the polygon, and
3. whether the winding number of the polygon relative to the midpoint of the potential diagonal is one.

As soon as a valid diagonal is found the triangulation algorithm normally proceeds with the two sub-polygons.



(a)



(b)

Figure 2: Ear-clipping for a self-overlapping polygon.

If we cannot find either ears or valid diagonals then we check whether there exists a self-intersection. That is, we check whether two edges (v_{i-1}, v_i) and (v_{i+1}, v_{i+2}) intersect in their relative interiors, cf. Fig. 2(b). If such an intersection exists then we locally cure the problem by first clipping the triangle v_i, v_{i+1}, v_{i+2} and then clipping the triangle v_{i-1}, v_i, v_{i+2}. Then, hopefully, the normal ear-clipping process can proceed.

We note that the insertion of those two partially overlapping triangles makes perfect sense when the polygon is viewed as part of a 3D surface. Simply inserting the point of intersection between (v_{i-1}, v_i) and (v_{i+1}, v_{i+2}) would not be a feasible solution, as the resulting 3D surface would have a gap. (Avoiding gaps is particularly important when triangulating the faces of 3D polyhedra.)

If no such self-intersection exists then the code finally enters "desperate mode". Since all other means have failed to make progress towards a meaningful triangulation the code randomly picks one convex vertex and clips the corresponding "ear". If no convex vertex exists then the code randomly picks a reflex vertex and clips it. It is important to point out that the code attempts to leave desperate mode as soon as possible. Indeed, we have already witnessed sample polygons for which the code proceeded with computing a meaningful triangulation, after it first got stuck at some twists and had to enter desperate mode.

Of course, the algorithm has no means to guarantee that a meaningful triangulation will be extracted out of a polygon that resembles a bale of hay. For such polygons the old principle "garbage in – garbage out" still applies. However, even for polygons that are messed up badly the algorithm will not get stuck or crash. Since we clip "ears" one way or the other, the topological structure of the dual of a triangulation computed by the code always is a tree, with the vertices of the triangulation being given by the vertices of the polygon. Also, every edge of the input polygon belongs to exactly one output triangle. (Thus, the triangulation computed is consistent at face boundaries, which again is important for triangulating the faces of 3D polyhedra or cells of "planar" subdivisions.) Obviously, the main crux is to generate a triangulation that is not only topologically correct but that also corresponds to a human's intuition of what a triangulation of a non-simple polygon should be.

We add that the bridge-finding algorithm described in Section 2.3 also is backed up by a desperate mode. For instance, we have to handle islands whose left-most point lies outside of the outer boundary polygon. Overlapping islands also cause problems for the standard bridge-finding algorithm. Similarly to the ear-clipping process, we gradually reduce the requirements for what constitutes a valid bridge. Ultimately, a line segment between two random vertices of different polygonal loops is accepted as a bridge if nothing better can be found.

Finally, we note that reliability does not come for free. The repeated attempts of the algorithm to avoid desperate mode may lead to a drastic increase of the cpu-time consumption: Basically, every deviation from the standard ear-clipping process means adding a cost of at least $O(n)$, and several $O(n)$ steps are carried out before the algorithm resorts to clipping one ear in desperate mode as its last means for handling a polygon. The decrease of efficiency will hardly be noticeable for small polygons, but repeated applications of these $O(n)$ steps may very well gobble up significant cpu resources for a complex polygon that has hundreds of twists and overlaps. (The actual clipping of an ear in desperate mode is computationally cheap, though.)

5 Experimental Results

5.1 Set-up of Experiments

A major goal of this work was to prove experimentally that this ear-clipping algorithm is not only reliable but also efficient. Thus, it seems to be natural to test the code on data arising in practice, and to compare it to codes that implement triangulation algorithms with better complexity bounds. Unfortunately, a sufficiently large amount of practical data is hard to come by. Also, practical data usually does not scale well. That is, due to idiosyncrasies of the test data it is somewhat problematic to make any educated guesses of the behavior of an algorithm unless it can be run on a very large amount of test data.

We opted for a different approach, based on synthetic test data. We generated 10 polygons each for sets of 2^i vertices, where $3 \leq i \leq 15$. (Thus, the numbers of vertices ranged between 8 and 32,768.) This procedure was repeated for four different classes of test polygons.

All polygons of the first three classes were generated by means of RandomPolygonGenerator (RPG), cf. [1, 2], which is a tool designed for the machine generation of pseudo-random polygonal test data. All polygons of the first class of test polygons, which was dubbed "random", were generated by using RPG's heuristic 2-opt Moves, applied to points uniformly distributed within the unit sphere. As discussed in [1, 2], this heuristic generates highly complex pseudo-random polygons. Our second class of test polygons, dubbed "smoother", was generated by smoothing the output of 2-opt Moves four times by means of RPG's algorithm Smooth. (Smooth produces a new simple polygon with twice the number of vertices by replacing vertex v_i of the polygon by the two new vertices $\frac{v_{i-1}+3v_i}{4}$ and $\frac{v_{i+1}+3v_i}{4}$. The result of a repeated application of Smooth is a fairly smooth polygon which closely resembles a straight-line approximation of a free-form curve.) Our third class of test polygons, called "smooth", was generated by applying Smooth only twice to the output of 2-opt Moves. The fourth class, called "thinned" polygons, was obtained by randomly clipping three quarters of the ears of the "random" polygons.

Fig. 3 depicts four polygons with 64 vertices, one for each of our four test classes. The difference in the characteristics of the polygons (and of one of their triangulations) is clearly visible. The "thinned" polygons tend to cover much less space than the "random" polygons do. Both the "thinned" and the "smoother" polygons have vertices that are (highly) non-uniformly distributed. Also, the "smoother" polygons tend to have very short edges, but quite a few comparatively long diagonals. Of course, the complexity of the polygons within each class increases as the number of vertices increases.

5.2 Experimental CPU-Time Consumption

The ear-clipping algorithm explained in this paper has been implemented in ANSI C. By means of compile-time switches we can select the different methods of geometric hashing used to speed up the ear-clipping process. The cpu-time consumption of our code, and of the other codes, is obtained by using the C system function "getrusage()". We report both the system and the user time. Of course, any file I/O and similar preprocessing is not included in the timings reported. All cpu times are given in milliseconds.

Besides our own code, named "FIST" as an acronym for "Fast Industrial-Strength Triangulation", we have tested the following C codes:

- Narkhede and Manocha's implementation [34, 35] of Seidel's algorithm [40];
- Saade's implementation [39] of Toussaint's algorithm [47];
- Shewchuk's "Triangle" [43, 41], which computes a constrained (resp., conforming) Delaunay triangulation; and
- Sloan's implementation [45] of an ear-clipping algorithm. (Sloan's code clips ears according to CE2. Its complexity is $O(n \cdot r)$ for triangulating an n-gon with r reflex vertices.)

All tests reported here were carried out on a Sun Ultra 1 running Solaris 2.5. (Our test machine had 320MB of main memory, but memory clearly was no issue for any of the

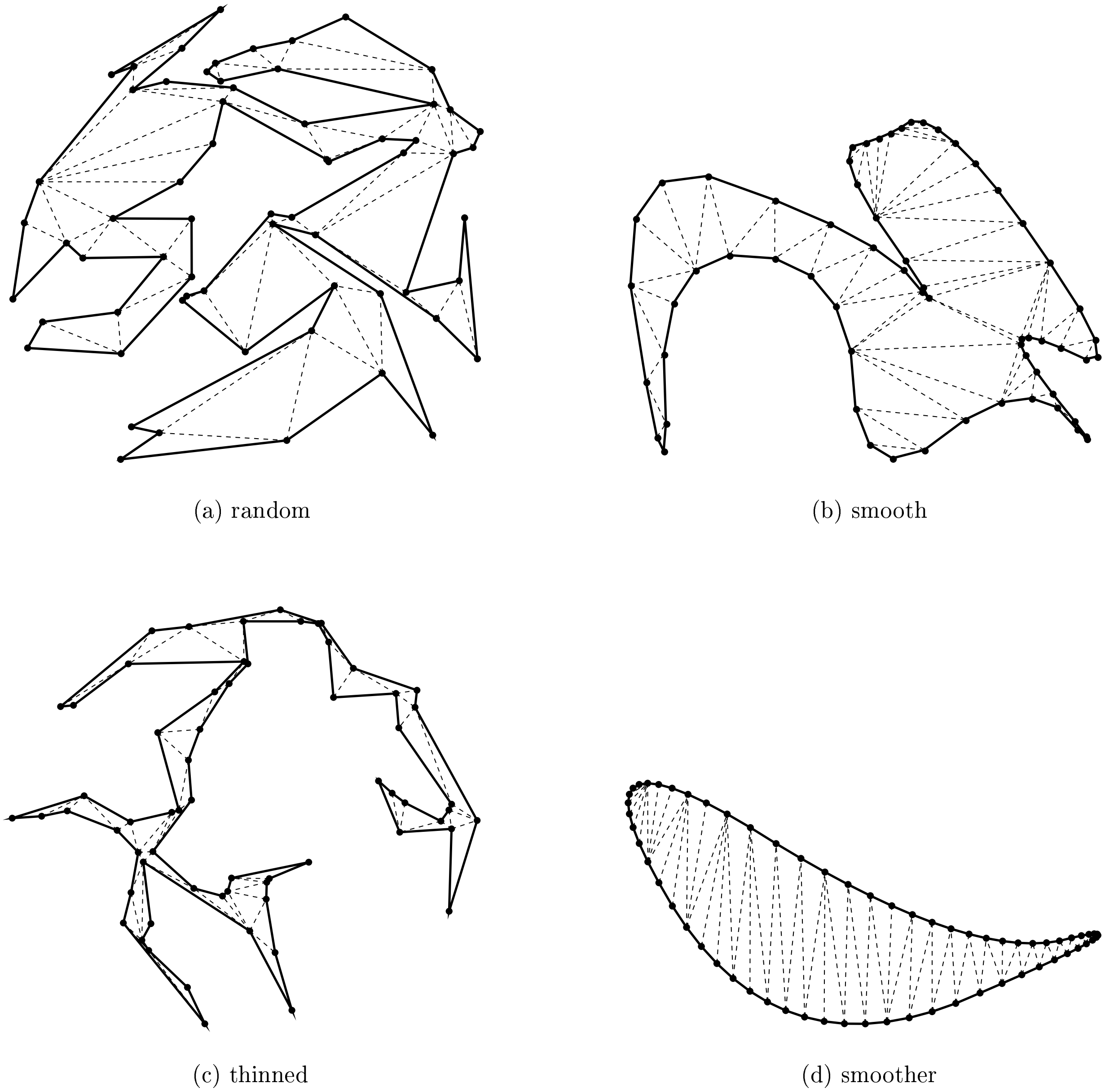
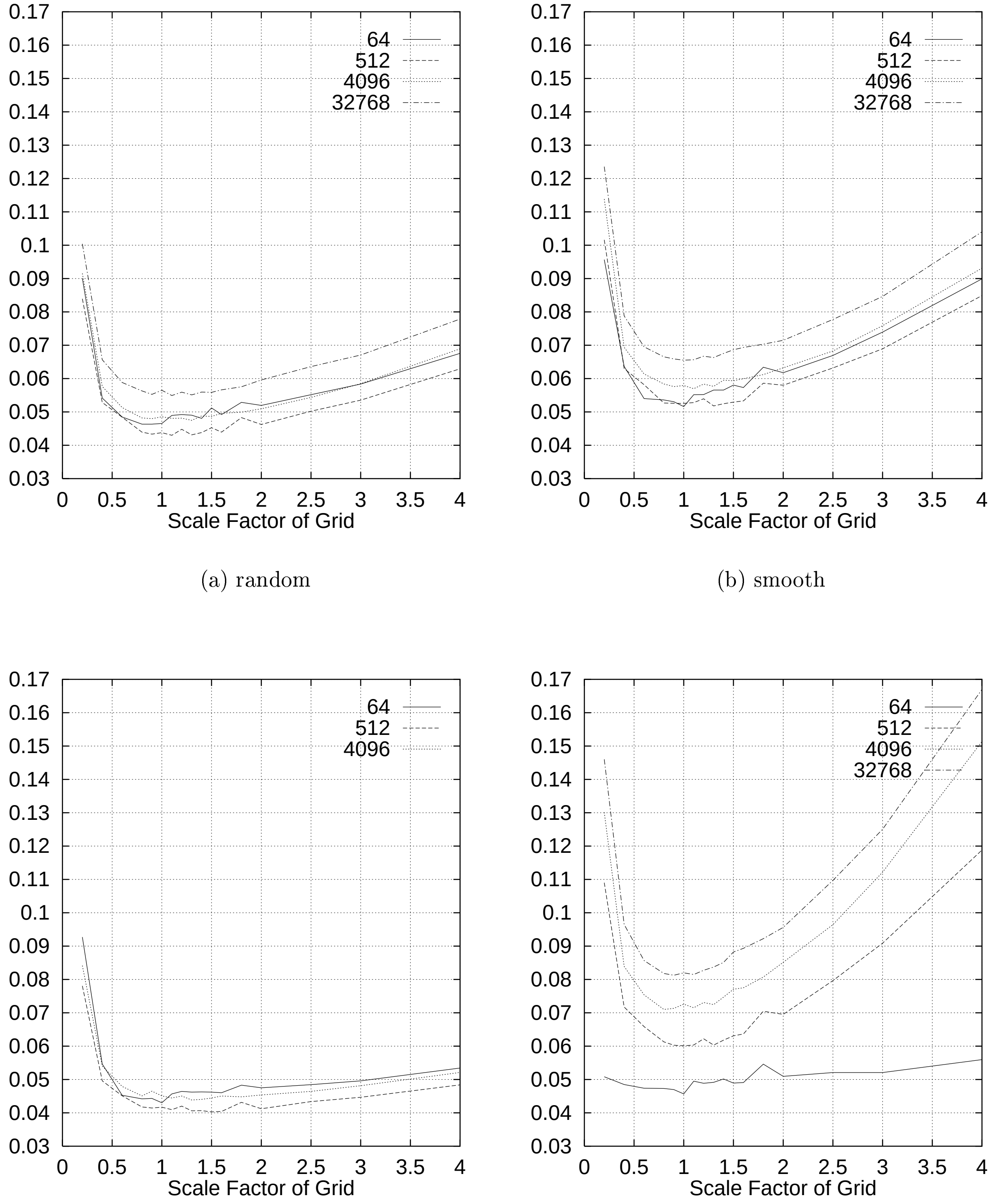


Figure 3: Sample 64-gons for the four classes of test polygons.

codes.) All codes were compiled with GNU's `gcc`, using the optimization level "-O2". The individual entries of the subsequent cpu-time charts were generated by averaging the cpu-time consumptions over each group of 10 polygons with the same number of vertices.

We started with experimentally determining the optimal resolution of the grid for clipping ears according to CE2. For each of the four classes of test polygons, Fig. 4 shows the cpu-time consumptions per segment for different values of $w \cdot h$, when using "random" ear clipping based on CE2. Each chart depicts timings of the code when applied to polygons with 64, 512, 4096, and 32768 vertices. As can be seen in Fig. 4, the best value for $w \cdot h$ is largely independent of the actual class of polygons or of the number of vertices within a class. Roughly, the code seems to perform best when $w \cdot h$ is about 1. Thus, the



(a) random (b) smooth

(c) thinned (d) smoother

Figure 4: CPU time (per segment) versus scale factor $w \cdot h$ of the grid, for grid-based clipping of ears according to CE2. The charts plot the cpu times for triangulating polygons with 64, 512, 4096, and 32768 vertices.

remaining tests for clipping ears according to CE2 are based on a grid size of $w\sqrt{n} \times h\sqrt{n}$, with $w \cdot h = 1$. Note that the cpu-time consumption is least sensitive to the grid size for "thinned" polygons, and most sensitive for "smoother" polygons. In any case, however, the cpu-time consumption varies only insignificantly for $0.5 \leq w \cdot h \leq 2$.

In a similar experiment, we have obtained $w \cdot h = 2.5$ as the best choice for the grid resolution when clipping ears according to CE1. Fig. 5 shows the cpu-time consumptions (per segment) of the different variants of geometric hashing used for speeding up the ear-clipping process. We have tested grid-based ear clipping according to CE1, CE1/grid, and according to CE2, CE2/grid. Grid-based ear clipping has been compared to clipping ears according to CE1 by means of a BV-tree, with continuous update of the tree, CE1/tree, and without a continuous update, CE1/tree w/o update.

All runs used "random" clipping of ears, and have been timed for polygons ranging from $2^3 = 8$ vertices to $2^{15} = 32768$ vertices. Clearly, the grid-based ear-clipping algorithms performed best, with ear clipping according to CE2 taking the win. When using a BV-tree, it is better to recompute the entire BV-tree from scratch once in a while, rather than to perform regular updates.

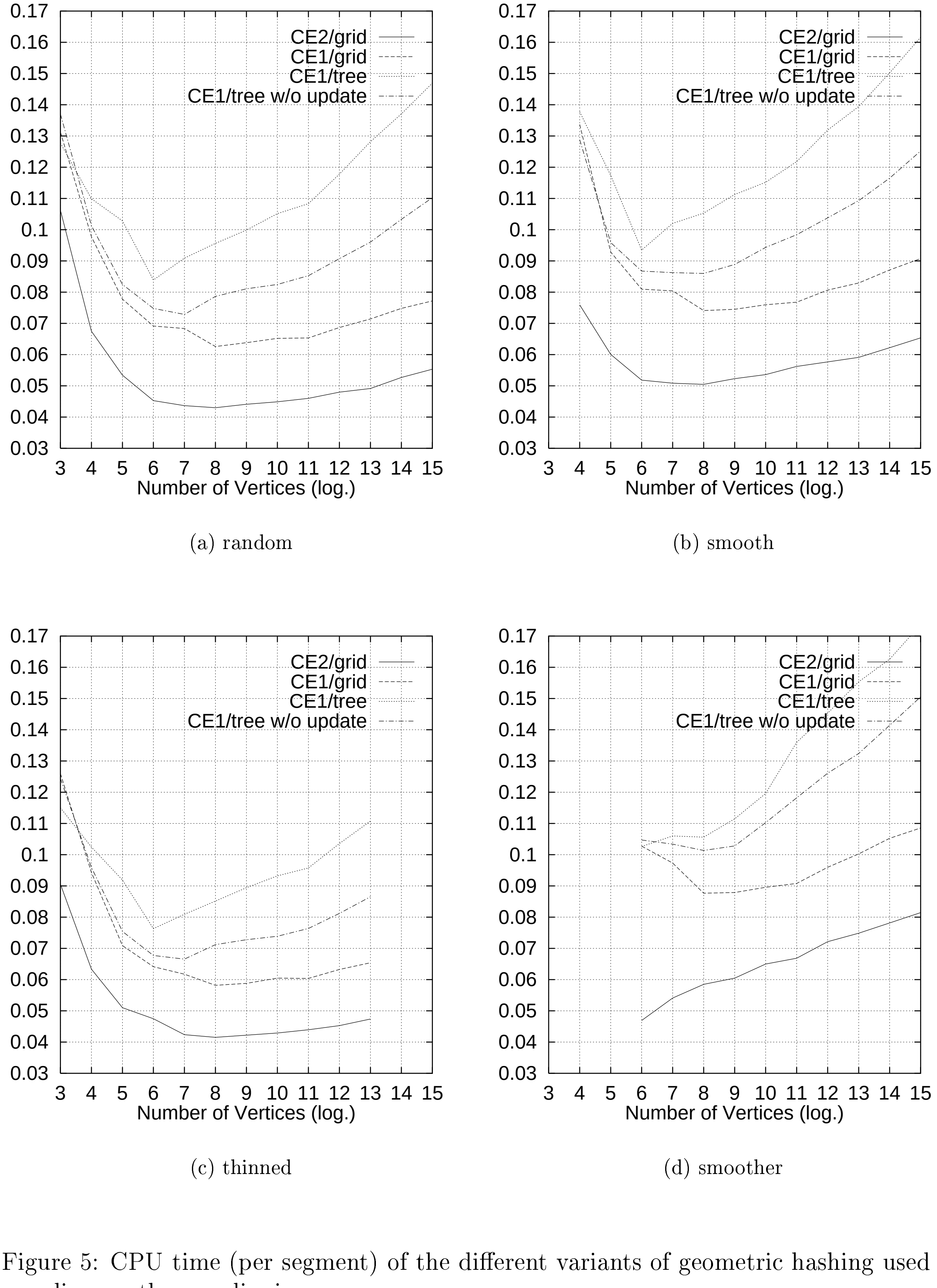
Finally, we turned our attention to a comparison of our fastest variant of the code, based on using a grid for clipping ears according to CE2, with the other triangulation codes listed above. Since our code is the only triangulation code[5] that can handle degeneracies or self-intersections we made sure that all test polygons were truly simple polygons. In Fig. 6, rnd denotes our "random" ear-clipping, sort stands for "sorted" ear-clipping, unc stands for Narkhede and Manocha's implementation [34, 35] of Seidel's algorithm [40], shew stands for Shewchuk's "Triangle" [43, 41], and sloan denotes Sloan's ear-clipping code [45].

Among our codes, the "sequential" ear clipping was about 5% faster than "random" ear clipping. Since "random" clipping yields much nicer looking triangulations we did not plot the cpu-time consumption of "sequential" ear clipping. "Sorted" ear-clipping typically is about 10-15% slower than "random" clipping. Our codes performed best on the "thinned" and "random" polygons, slightly worse on the "smooth" polygons, and worst on the "smoother" polygons. Clearly, geometric hashing did a poor job of weeding out reflex vertices as the vertices ended up in a small number of the cells, while quite a few diagonals spanned several cells of the grid.

The "smoother" polygons were the easiest to process for Shewchuk's "Triangle" and for Narkhede and Manocha's implementation of Seidel's algorithm. Both performed best for the "smoother" polygons, and worst for the "thinned" polygons. However, the variation of the cpu-time consumption of those two codes was less visible among the different classes of polygons, as could be expected. Compared to our code, those two codes were noticeably slower for "random" and "thinner" polygons, whereas the differences became less noticeable for some of the "smooth(er)" polygons. Actually, Shewchuk's "Triangle", shew, took the global win for "smoother" polygons with 256 vertices.

Also, as expected, Shewchuk's "Triangle" and Narkhede and Manocha's implementation of Seidel's algorithm showed only a slightly super-linear increase in cpu-time consumption, as did our code. Since we sort the input vertices and use binary search, we cannot hope to

[5] Shewchuk's "Triangle" handles self-intersections by inserting the point of intersection as a Steiner point. Thus, it computes a triangulation that conforms to the arrangement of the input segments, and then deletes those triangles that lie in the exterior of the polygon. As clearly stated, his code may keep the wrong part of the triangulation in the presence of identical vertices. The other codes tested die gracefully (in the best case) or simply dump core when applied to polygons that are not truly simple.



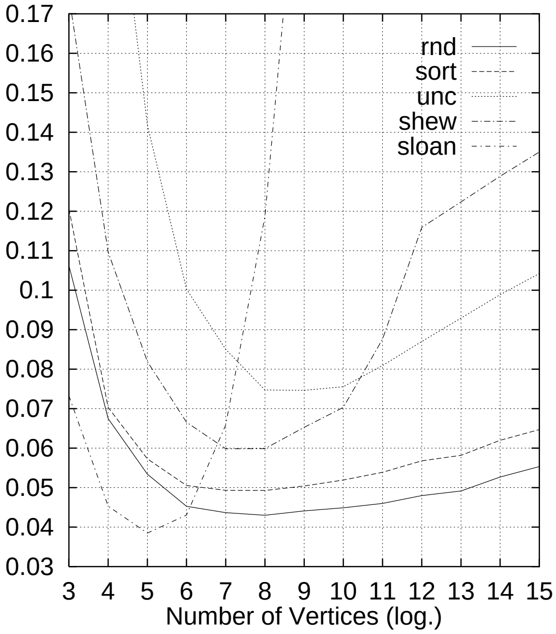
(a) random

(b) smooth

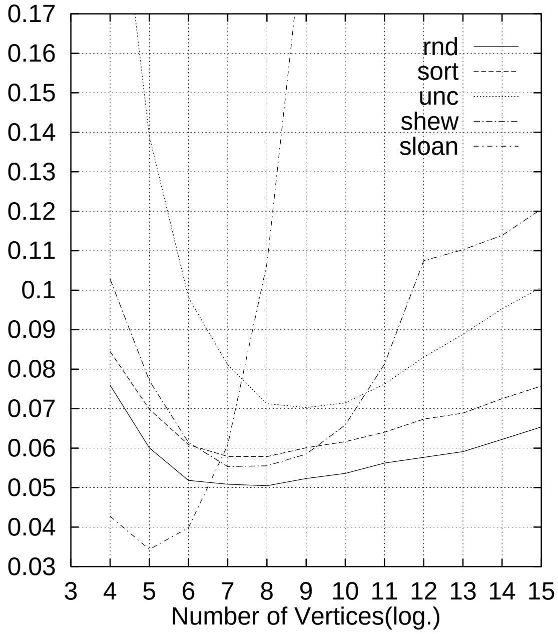
(c) thinned

(d) smoother

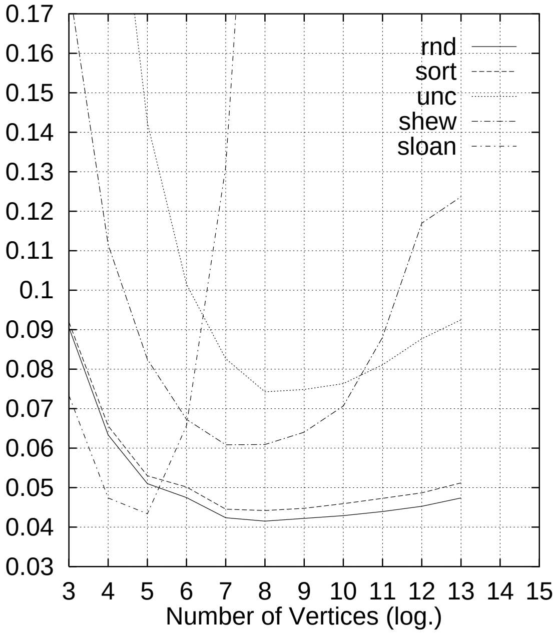
Figure 5: CPU time (per segment) of the different variants of geometric hashing used for speeding up the ear-clipping process.



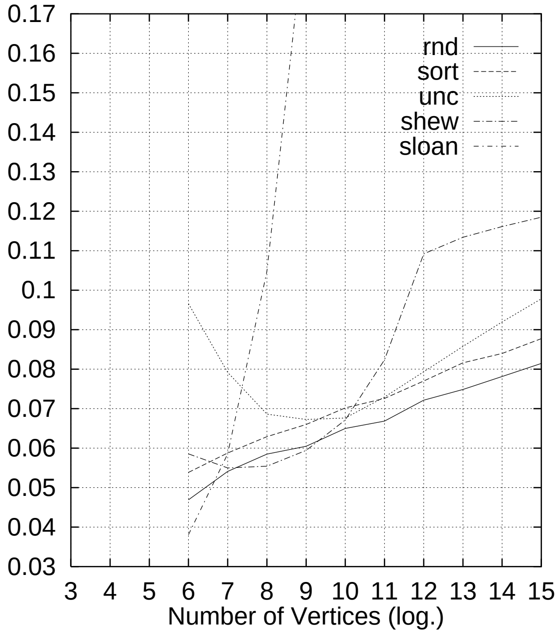
(a) random



(b) smooth



(c) thinned



(d) smoother

Figure 6: CPU time (per segment) for the triangulation codes. We have tested “random” (rnd) and “sorted” (sort) ear clipping, and compared it to Narkhede and Manocha’s implementation of Seidel’s algorithm (unc), Shewchuk’s code (shew), and Sloan’s code (sloan).

observe a purely linear cpu-time consumption. However, the charts seem to suggest that our code is the least effected by an increase in the input size, for any of the four classes of test polygons.

5.3 Does Geometric Hashing Pay Off?

The tests reported in the previous subsection indicate that geometric hashing pays off nicely for large polygons. But what is the price that we have to pay when applying geometric hashing to small polygons? For instance, using a grid for triangulating a 10-gon might constitute an unjustified overhead, even when using only a correspondingly small 3×3 grid. And how poorly would the code perform if no hashing were used at all? In order to answer these questions we introduced another compile-time option that allows us to use no hashing at all. Rather, all reflex vertices are maintained in a list, and a point-in-triangle test is carried out for all vertices contained in this list in order to perform an earity test. In order to weed out vertices that cannot lie inside a triangle we employ a bounding-box pre-test. Also, vertices that are no longer reflex get deleted from this list.

In Fig. 7, rnd (hash) denotes grid-based "random" clipping of ears according to CE2, while rnd (no hash) denotes the "random" clipping of ears without the use of a grid, as described above. For comparison purposes we plotted the cpu-time consumption of Sloan's ear-clipping code [45], sloan, and of Saade's implementation [39] of Toussaint's algorithm [47], saade. For obvious reasons we ran this experiment for no polygon with more than $2^{13} = 8192$ vertices.

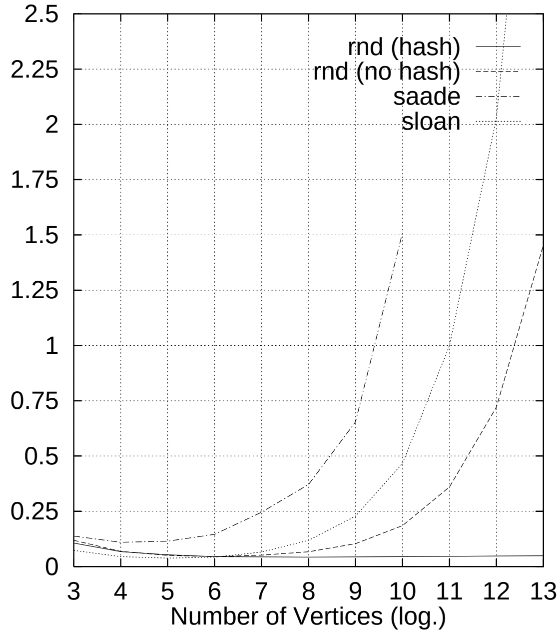
Sloan's code is the fastest polygon triangulator for small polygons with, say, less than $2^6 = 64$ vertices. For more complex polygons, however, it exhibits a truly $O(n^2)$ behavior and soon is noticeably slower than rnd (hash). Saade's implementation of Toussaint's algorithm performed similarly to Sloan's code, albeit with a higher constant (except for "thin" polygons). Its start-up costs were drastically higher, and it was not competitive for most of the tested polygons. (We should point out that Saade's code is not publicly available, and that we have explicitly been warned that it had not been optimized at all. We expect an optimized version to perform better, at least for small polygons.)

For polygons with up to about $2^6 = 64$ vertices there is very little difference in the speed of our grid-based version, rnd (hash), versus rnd (no hash) which does not use any form of hashing. It seems that in our implementation there is nothing to gain or lose by using a grid for small polygons. For larger polygons rnd (no hash) also exhibits a quadratic time complexity. It remains faster than Sloan's code, though.

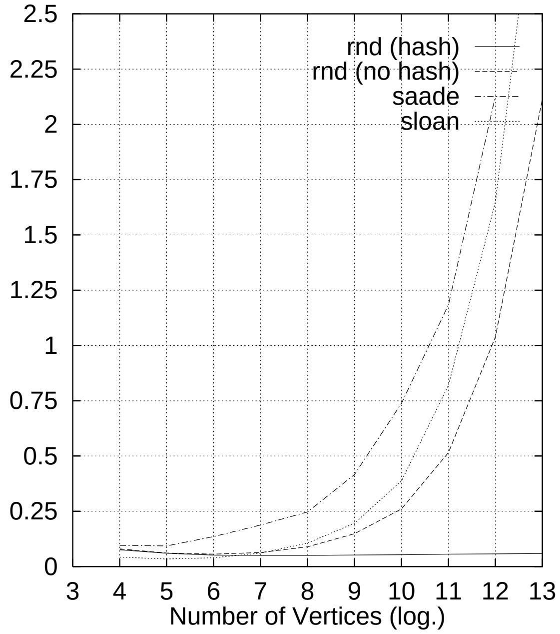
We tried to reduce the start-up costs of our code, in an attempt to speed it up for very small polygons. However, it became apparent that our implementation of the numerical primitives has to pay a price for improved reliability. We tried to mirror Sloan's approach as closely as possible within our computational framework. However, this only helped to reduce the cpu-time consumption by about 5% for very small polygons, at the expense of a more significant increase for larger polygons.

5.4 Caveats

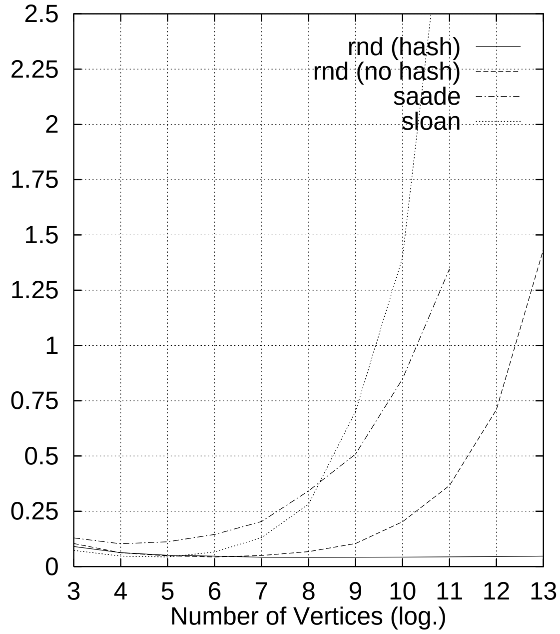
One may wonder whether there do exist polygons for which geometric hashing fails to avoid a quadratic time complexity. Our tests seemed to indicate that the distribution of the vertices of a polygon has little influence on the behavior of our code. Rather, using



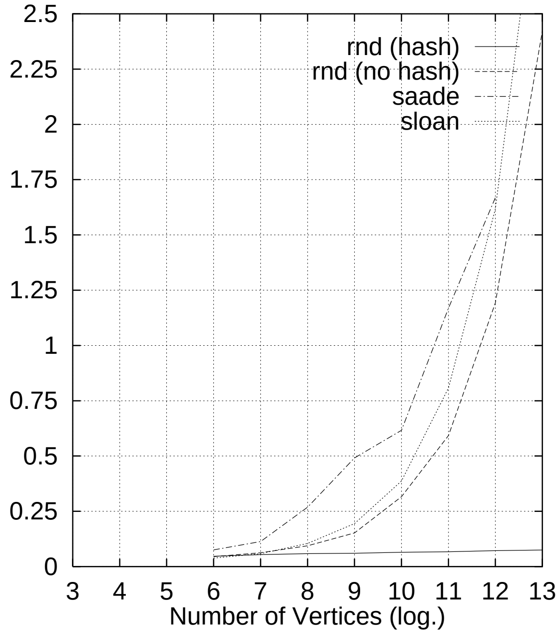
(a) random



(b) smooth



(c) thinned



(d) smoother

Figure 7: CPU time (per segment) for the $O(n^2)$ triangulation codes in comparison to our grid-based code, rnd (hash). Our ear clipping without hashing, rnd (no hash), is compared to Sloan's code, sloan, and Saade's implementation of Toussaint's algorithm, saade.

geometric hashing for ear clipping according to CE2 seems to be much more vulnerable to the long triangulation edges of the "smoother" data sets. Long edges cause ears with fairly large bounding boxes (relative to the size of the polygon), and thus require the testing of many more reflex vertices. In order to substantiate this hypothesis we decided to test the triangulation codes on star-shaped polygons. (Of course, star-shaped polygons can easily be triangulated in linear time, but none of the general-purpose triangulation codes exploits this feature.) Our star-shaped polygons had vertices distributed uniformly in the unit circle, but had lots of long edges which could be expected to yield ears with large bounding boxes. And, they had lots of reflex vertices.

Star-Shaped Polygons			
#(vertices)	512	4096	32768
rnd	0.066	0.191	1.250
shew	0.076	0.171	0.358
unc	0.078	0.093	0.112
sloan	0.197	1.642	13.56

Table 1: CPU time (per segment) for the different triangulation codes applied to star-shaped polygons with 512, 4096, and 32768 vertices.

Table 1 lists the cpu-time consumption of our grid-based "random" ear clipping according to CE2, rnd, in comparison to Shewchuk's code [43, 41], shew, and Narkhede and Manocha's implementation [34, 35] of Seidel's algorithm [40], unc. Sloan's code [45], sloan, is included for comparison purposes, too. While being a bit faster than the other codes for star-shaped $2^9 = 512$-gons, rnd is considerably slower than Shewchuk's code and Narkhede and Manocha's implementation of Seidel's algorithm for star-shaped polygons with $2^{15} = 32768$ vertices. Although the cpu-time consumption of rnd for star-shaped polygons still is no disaster, it clearly shows that geometric hashing does not always help to achieve a slightly super-linear complexity. (However, our grid-based code still is an order of magnitude faster than without using any hashing; compare the timings for rnd with the timings for Sloan's code sloan.)

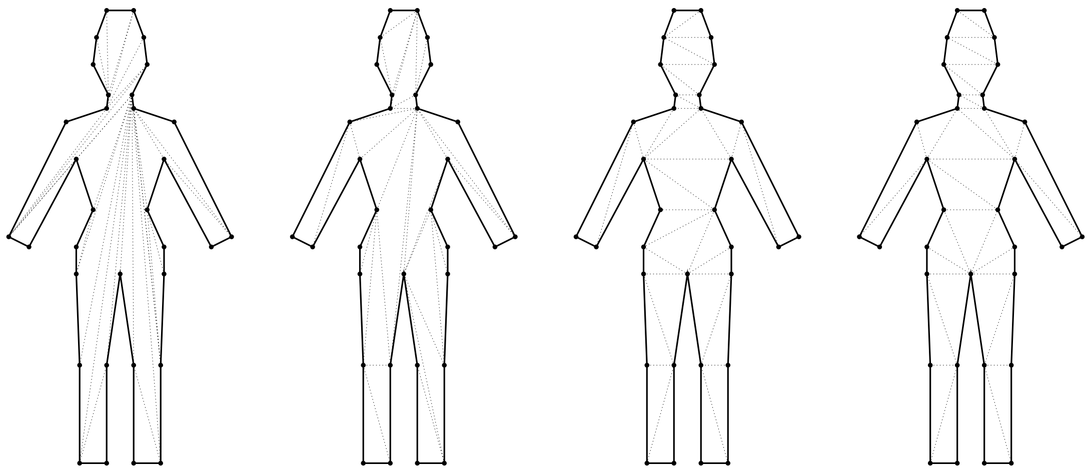
Also, note that Shewchuk's code performs best for polygons for which the Delaunay triangulation of the vertices (nearly) conforms to the edges of the polygon, such as for our "smoother" polygons, while it struggles considerably more if the Delaunay triangulation of the vertices of the polygon is far from conforming to the edges of the polygon. (It is about three times as slow for large star-shaped polygons as for comparable "smoother" polygons.)

Needless to say that we have experienced minor differences in the relative speeds of the codes when running them on a different platform. For instance, our SGI Indigo 2 (with an R10000 processor) seemed to favor pure number crunching over maintaining data structures and traversing complex function hierarchies. That is, the brute-force $O(n^2)$ codes did somewhat better on this SGI than on our Sun. However, the overall picture remained the same.

Finally, the reader is cautioned that any comparison of algorithms implemented by different programmers has to be taken with a grain of salt. Minor differences in the cpu-time consumption may be caused by different programming styles, or different efforts placed on optimizing a code. (And we have been told that Saade's code is not optimized at all.) However, since Shewchuk's and Narkhede and Manocha's codes are publicly available we feel that it is justified to assume that their codes also are optimized, at least to some extent.

5.5 Quality of the Triangulation

In some applications it is essential that the minimum interior angle of a triangle of the computed triangulation is as large as possible. While it never was an objective of this work to produce guaranteed-quality triangulations, we were startled to see that some of the "sequential" triangulations (`seq`) looked really bad, having lots of sliver triangles.



(a) seq (b) rnd (c) sort (d) shew

Figure 8: Sample triangulations computed by the different methods.

It soon became clear that "random" or "sorted" clipping of ears is a simple means to improve the quality of the triangulation, at the expense of a slightly increased cpu-time consumption. This visual observation was also supported by experimental analysis, for which we plotted the distribution of the minimum angles of the triangles for the different methods. Detailed charts are omitted; the characteristics of the different methods can be seen in Fig. 8.

We experimented with boosting "sorted" ear-clipping to "fancy" ear-clipping by trying to avoid to clip an ear if the diagonal would be close to another reflex vertex. (In this case, we would likely have to clip a sliver triangle later on.) While "fancy" ear clipping helped to improve the quality of the triangulation, it also caused the cpu-time consumption to increase by some additional 20 to 30%, thus defeating its purpose somewhat.

Summarizing, it became clear that any form of ear clipping in general produces a triangulation of inferior quality than the Delaunay triangulation, as generated by Shewchuk's "Triangle", no matter which add-on heuristic is used. Our "sorted" clipping of the ears normally produces nice looking triangulations, but it is bound to contain a few (unnecessary) sliver triangles, too. The only alternative to a Delaunay-triangulation based triangulator may be to perform edge flips as an optimization step after the triangulation has been computed, as offered by Sloan's code. However, when performing edge flips care has to be taken in order not to destroy the validity of the triangulation in the case of non-simple polygonal input data.

6 Practical Reliability Tests

6.1 Robustness without Desperate Mode

FIST relies on multi-level back-up procedures, which get applied if the ear-clipping algorithm runs out of ears, in order to prevent it from crashing. Thus, FIST cannot crash[6] and will always terminate with a topologically valid triangulation. The experience gained from applying FIST to real-world and synthetic data certainly concurred with its predicted reliability: we are yet to see it crash. Thus, the multi-level back-up procedures, which ultimately lead to desperate mode, seem to work in theory and in practice.

However, we note that the availability of such back-up procedures can easily conceal bugs in the core of the triangulation code. Obviously, FIST should not end up in desperate mode when dealing with a truly simple or degenerate polygon. In order to check its robustness we disabled all the back-up procedures. Then, we re-ran the tests for all our random data sets, and applied FIST to many more degenerate polygons. FIST finished successfully for all those data sets, with what seemed to be correct triangulations. Therefore, we are confident that FIST will resort to its back-up procedures (and the final desperate mode) only for truly non-simple polygons. (Of course, even such an extensive test can only show the existence of a bug, but does not help to prove that there is no bug lurking somewhere in the code.)

6.2 Tessellation of Real-World Polyhedral Faces

We have extended FIST to be able to parse descriptions of 3D polyhedral models encoded in Wavefront's `.obj` format. For every polyhedral face a plane that approximately contains the vertices of the face is computed. The normal vector of this approximating plane is obtained by averaging over all (unit) normal vectors defined by triples of consecutive vertices. After an approximating plane has been established, the vertices of the face are projected onto this plane, and the resulting 2D polygon is then triangulated by our triangulation code. The triangles of the resulting triangulation of the surface of the polyhedron can then be grouped into quadrangles. (Grouping triangles into quadrangles is a cheap way for doing a very simple geometric data compression.)

Of course, triangles and quadrangles form the majority of faces of real-world polyhedra, and faces with few vertices are not subjected to the general triangulation procedure but are dealt with directly. However, more complex faces do occur, and it was interesting to see that perhaps some dozen out of a few million of polyhedral faces actually caused FIST to end up in desperate mode. Likely, the code would have crashed even more often if we had not carefully implemented the handling of degenerate polygons.

FIST has already survived the acid test of being applied to real-world data within an industrial environment. Our colleagues at Sun Microsystems have integrated it into two industrial graphics packages: FIST has become part of "HoloSketch" and of "Leotool".

[6] Well, unless there were a bug in the actual implementation ...

Currently, FIST gets translated to Java 3D, in order to be integrated into Sun's implementation of Java 3D.

7 Conclusion

We presented a triangulation algorithm based on ear clipping. Information on the reflex vertices of a polygon is maintained in a regular grid, and this grid-based geometric hashing is used to speed up the earity checks. Extensive experimental tests have shown that this grid-based geometric hashing works well in order to make this approach competitive with (or even faster than) other triangulation codes which have a better worst-case complexity, such as Narkhede and Manocha's implementation of Seidel's algorithm and Shewchuk's "Triangle". As could be expected, geometric hashing has little effect for small polygons. In our implementation the differences between using and not using geometric hashing are negligible for polygons with up to about 64 vertices.

Particular care has been taken in order to make the algorithm reliable. Due to a series of heuristics that serve as a back-up for the standard ear-clipping process, our triangulation code ("FIST") will never crash when applied to polygonal data which has deficiencies. Rather, it will always attempt to generate a meaningful triangulation, and the intuitive "correctness" of the triangulation will gracefully degrade the more the polygon is messed up.

Acknowledgments

This work has benefited from discussions with Joe Mitchell and his students at SUNY Stony Brook, and with Henry Sowizral and Karel Zikan and their colleagues at Sun Microsystems. Godfried Toussaint pointed us to interesting literature. The implementation of Toussaint's algorithm was graciously provided by Elie Saade. Our experiments also involved triangulation codes made publicly available by Dinesh Manocha, Atul Narkhede, Jonathan Shewchuk, and Kenneth Sloan.

References

[1] 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.

[2] T. Auer and M. Held. RPG – Heuristics for the Generation of Random Polygons. *Submitted for publication*, 1998.

[3] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluating Signs of Determinants Using Single-Precision Arithmetic. *Algorithmica*, 17:111–132, 1997.

[4] R. Balbes and J. Siegel. A Robust Method for Calculating the Simplicity and Orientation of Planar Polygons. *Comput. Aided Geom. Design*, 8(4):327–335, Oct 1991.

[5] R. Bar-Yehuda and B. Chazelle. Triangulating Disjoint Jordan Chains. *Internat. J. Comput. Geom. Appl.*, 4(4):475–481, 1994.

[6] G. Barequet, M. Dickerson, and D. Eppstein. On Triangulating Three-Dimensional Polygons. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 38–47, Philadelphia, PA, USA, June 1996.

[7] G. Barequet and Y. Kaplan. A Data Front-End for Layered Manufacturing. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 231–239, Nice, France, June 1997.

[8] G. Barequet and M. Sharir. Piecewise-Linear Interpolation between Polygonal Slices. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 93–102, Stony Brook, NY, USA, 1994.

[9] G. Barequet and M. Sharir. Filling Gaps in the Boundary of a Polyhedron. *Comput. Aided Geom. Design*, 12(1):207–229, 1995.

[10] G. Barquet and B. Wolfers. Optimizing a Corridor Between Two Polygons with an Application to Polyhedral Interpolation. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 32–37, Ottawa, Canada, Aug 1996. Carleton University Press.

[11] M. Bern and D. Eppstein. Mesh Generation and Optimal Triangulation. In D.-Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, pages 23–90. World Scientific, 1992. ISBN 981-02-0966-5.

[12] H. Brönnimann, I. Emiris, V. Pan, and S. Pion. Computing Exact Geometric Predicates Using Modular Arithmetic with Single Precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, Nice, France, June 1997.

[13] H. Brönnimann and M. Yvinec. Efficient Exact Evaluation of Signs of Determinants. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 166–173, Nice, France, June 1997.

[14] C. Burnikel et al. Exact geometric Computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, Vancouver, BC, Canada, 1995.

[15] B. Chazelle. A Theorem on Polygon Cutting with Applications. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 339–349, 1982.

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

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

[18] Y.-K. Choi and K.H. Park. A Heuristic Triangulation Algorithm for Multiple Planar Contours Using an Extended Double Branching Procedure. *Visual Comput.*, 10(7):372–387, 1994.

[19] K.L. Clarkson, R.E. Tarjan, and C.J. Van Wyk. A Fast Las Vegas Algorithm for Triangulating a Simple Polygon. *Discrete Comput. Geom.*, 4:423–423, 1989.

[20] O. Devillers. Randomization Yields Simple $O(n \log^\star n)$ Algorithms for Difficult $\Omega(n)$ Problems. *Internat. J. Comput. Geom. Appl.*, 2(1):97–111, Mar 1992.

[21] H. Edelsbrunner and E.P. Mücke. Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms. *ACM Trans. Graph.*, 9(1):66–104, Jan 1990.

[22] H. ElGindy, H. Everett, and G. Toussaint. Slicing an Ear Using Prune-and-Search. *Pattern Recogn. Lett.*, 14:719–722, 1993.

[23] A. Fabri et al. The CGAL Kernel: A Basis for Geometric Computation. In *1st ACM Workshop Appl. Comput. Geom.*, pages 97–103, Philadelphia, PA, USA, May 1996.

[24] A. Fournier and D.Y. Montuno. Triangulating Simple Polygons and Equivalent Problems. *ACM Trans. Graph.*, 3(2):153–174, 1984.

[25] M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan. Triangulating A Simple Polygon. *Inform. Process. Lett.*, 7:175–179, 1978.

[26] M. Held, J.T. Klosowski, and J.S.B. Mitchell. QuickCD: An Efficient Collision Detection System Using BV-Trees. Manuscript, Dept. of Appl. Math. Stat., SUNY at Stony Brook, NY, USA, Mar 1997.

[27] URL: `http://www.cosy.sbg.ac.at/~held/projects/triang/triang.html`.

[28] S. Hertel and K. Mehlhorn. Fast Triangulation of Simple Polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, volume 158 of *Lecture Notes Comput. Sci.*, pages 207–218. Springer-Verlag, 1983.

[29] D.G. Kirkpatrick, M.M. Klawe, and R.E. Tarjan. Polygon Triangulation in $O(n \log \log n)$ Time With Simple Data Structures. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 34–43, Berkeley, CA, USA, 1990.

[30] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Trans. Visualizat. Comput. Graph.*, To appear 1997.

[31] X. Kong, H. Everett, and G. Toussaint. The Graham Scan Triangulates Simple Polygons. *Pattern Recogn. Lett.*, 11:713–716, Nov 1991.

[32] K. Mehlhorn and S. Näher. LEDA: A Platform for Combinatorial and Geometric Computing. *C. ACM*, 38(1):96–102, Jan 1995.

[33] G.H. Meisters. Polygons have Ears. *Amer. Math. Monthly*, 82:648–651, 1975.

[34] A. Narkhede and D. Manocha. Fast Polygon Triangulation Based on Seidel's Algorithm. In A.W. Paeth, editor, *Graphics Gems V*, pages 394–397. Academic Press, 1995. ISBN 0-12-543455-3.

[35] URL: `http://www.cs.unc.edu/~dm/CODE/GEM/chapter.html`.

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

[37] M.H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In *1st ACM Workshop Appl. Comput. Geom.*, pages 113–119, Philadelphia, PA, USA, May 1996.

[38] R.P. Ronfard and J.R. Rossignac. Triangulating Multiply-Connected Polygons: A Simple, Yet Efficient Algorithm. *Comput. Graph. Forum*, 13(3):281–292, 1994. (Proc. Eurographics'94).

[39] E. Saade. Personal Communication, 1997.

[40] R. Seidel. A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons. *Comput. Geom. Theory and Appl.*, 1(1):51–64, July 1991.

[41] URL: `http://www.cs.cmu.edu/~quake/triangle.html`.

[42] J.R. Shewchuk. Robust Adaptive Floating-Point Geometric Predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, Philadelphia, PA, USA, June 1996.

[43] J.R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *1st ACM Workshop Appl. Comput. Geom.*, pages 124–133, Philadelphia, PA, USA, May 1996.

[44] C. Silva, J.S.B. Mitchell, and A.E. Kaufman. Automatic Generation of Triangular Irregular Networks Using Greedy Cuts. In *Visualization '95*, pages 201–208, San Jose, CA, USA, 1995. IEEE Comput. Society Press.

[45] URL: `ftp://ftp.cis.uab.edu/pub/sloan/Software/triangulation/src/`.

[46] R.E. Tarjan and C.J. Van Wyk. An $O(n \log \log n)$-time Algorithm for Triangulating a Simple Polygon. *SIAM J. Comput.*, 17:143–178, 1988.

[47] G.T. Toussaint. Efficient Triangulation of Simple Polygons. *Visual Comput.*, 7(5–6):280–295, Sep 1991.

[48] C.K. Yap. Robust Geometric Computation. In J.E. Goodman and J. O'Rourke, editors, *CRC Handbook of Discrete and Computational Geometry*, pages 653–668. CRC Press, 1997. ISBN ISBN 0-8493-8524-5.

[49] C.K. Yap and T. Dubé. The Exact Computation Paradigm. In D.-Z. Du and F.K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific, 1995. ISBN 981-02-1876-1.