Parallel Triangulation of Polygons

Masterarbeit

zur Erlangung des akademischen Grades
Diplom-Ingenieur
an der Naturwissenschaftlichen Fakultät
der Paris Lodron Universität Salzburg

Eingereicht von Günther Eder

Gutachter: Ao. Univ.-Prof. Dipl.-Ing. Dr. Martin Held
Fachbereich Computerwissenschaften

Salzburg, Juli 2014

Parallel Triangulation of Polygons

Master's Thesis

Günther Eder

July 2014

Department of Computer Sciences
University of Salzburg
Jakob-Haringer-Straße 2
5020 Salzburg
Austria

Günther Eder

[July 11, 2014]

ABSTRACT

In this work we review five different triangulation algorithms and present two of our own. First, two well known algorithms are surveyed: ear-clipping and monotone subdivision. Then, three constrained Delaunay triangulation algorithms are discussed in detail: the first uses a Fortune-like sweep-line approach, the second uses a randomized incremental construction method, and the third is constructing the triangulation in parallel on the GPU.

We also present our two parallel ear-clipping methods. One is a divide and conquer approach where the simple input polygon is divided in linear time. The other is a mark and cut extension which uses a sequential mark phase and a parallel cut phase. Both are tested extensively and the results are discussed.

DEDICATION

To my family.

ACKNOWLEDGMENTS

Thanks to my advisor Martin Held for the support and the lectures in this interesting field. Also thanks to my colleagues for their time for discussions on this subject. Thanks to my family for their support.

TABLE OF CONTENTS

1 INTRODUCTION

In this chapter we declare definitions and state why polygon triangulation is still an important topic. We discuss who was involved in the research of this field and also the contribution of this work.

1.1 DEFINITION

We state some basic definitions and lemmas which are used throughout this work. Since most of the lemmas are well known, we do not provide proofs for each one.

DEFINITION 1 - POLYGON
Let $v_1, v_2, ..., v_n$ be vertices in the plane. Let $e_i := \overline{v_i v_{i+1}}$, with $1 \leq i \leq n$ and $e_n := \overline{v_n v_1}$, be consecutive edges which form a closed chain. Then P is a polygon consisting of the closed contour loop $e_1, e_2, ..., e_n$.

DEFINITION 2 - SIMPLE POLYGON
A polygon is called simple if exactly two edges meet at each vertex and two edges only intersect at a common endpoint.

In literature the interior region of a polygon P is also called its *body* or is referred to by the use of the interior function $Int(P)$.

DEFINITION 3 - PSLG
A planar straight line graph (PSLG) is the embedding of a planar graph in the plane where each edge is represented by a straight-line segment.

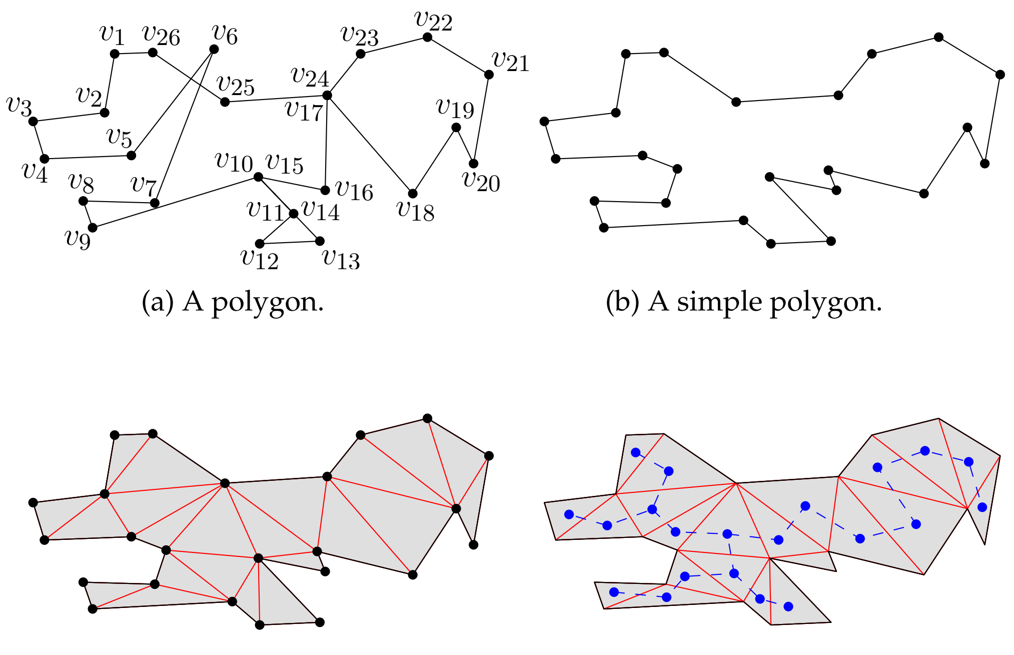
DEFINITION 4 - DIAGONAL [HEL01]
Let P be a polygon and $v_1, ..., v_n$ vertices which form the consecutive

contour of P. Then the line segment $\overline{v_i v_j}$ forms a diagonal of a P if $\overline{v_i v_j}$ lies completely in the interior of P, except for v_i and v_j.

DEFINITION 5 - TRIANGULATION [BCKO08]
A decomposition of a polygon P into triangles by a maximal set of non-intersecting diagonals is called a triangulation of P.

LEMMA 1 - TRIANGULATION ALWAYS EXISTS
Every polygon can be triangulated.

A proof for Lemma 1 can be found in the famous article *Polygons Have Ears* by Meisters from 1975 [Mei75].



(a) A polygon.

(b) A simple polygon.

(c) A triangulation of (b).

(d) The dual graph of (c).

Figure 1: An example of a non-simple polygon (a), a simple polygon (b), a triangulation of that simple polygon (c) and the dual graph of that triangulation (d).

LEMMA 2 - NUMBER OF TRIANGLES
Any triangulation T of a simple polygon P with n vertices consists of exactly $n - 2$ triangles.

LEMMA 3 The dual graph of a triangulation is a tree, with maximum degree of three.

There is no known tight bound for the maximum or minimum number of different triangulations of a point set with n vertices in 2D. A set of 6 vertices in convex position already admits 14 different triangulations. In 2009, Sharir, Sheffer and Welzl showed that there are at most 30^n different triangulations for n vertices in the plane [SSW09].

In Figure 1 we provide visualizations of the above definitions and lemmas.

1.1.1 *Delaunay Triangulation*

It is common to consider a "quality" measure for triangles. One quality of a triangle can be seen as its minimum internal angle. Maximizing the minimum internal angle helps to avoid skinny triangles (sliver triangles). In a practical application like the finite element method such triangles lead to an increased number of iterations needed for the computation.

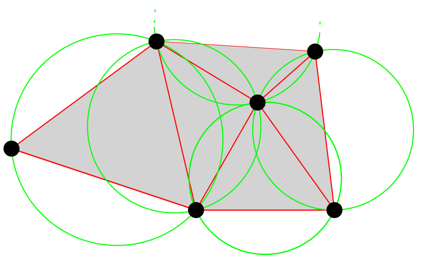
The Delaunay triangulation (DT), named after Boris Nikolaevich Delaunay (1890-1980) [Del34], is a triangulation of a point set on the plane. The DT is an optimal triangulation where the optimality is given by the maximization of the smallest internal angles over all triangles among all possible triangulations.

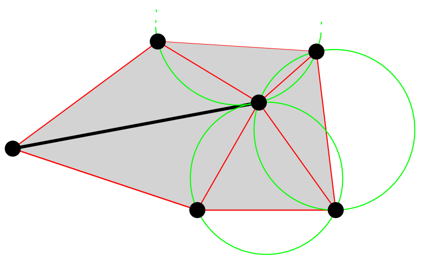
Various definitions for the DT are known. One can define the DT by the use of the *empty circle* property. It means that the circumcircle defined by the vertices of a triangle never contains further vertices. Another, more precise definition is given below.

DEFINITION 6 - DELAUNAY TRIANGULATION [CHE89]
Let S be a set of points in the plane. A triangulation T is a Delaunay triangulation of S if for each edge e of T there exists a disc d with the following properties:

1. The endpoints of edge *e* are on the boundary of *d*.
2. No other vertex of *S* is in the interior of *d*.

LEMMA 4 - UNIQUENESS
If no four points of *S* are co-circular then the Delaunay triangulation of *S* is unique.


(a) A Delaunay triangulation.


(b) A constrained DT.

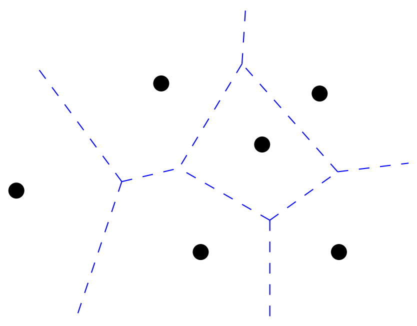

(c) A Voronoi diagram.

Figure 2: An example of a Delaunay triangulation of a point set S (a), a constrained Delaunay triangulation (b), and a Voronoi diagram of the same point set S (c).

The Delaunay triangulation (DT) is the dual graph of the Voronoi diagram (VD). Since various approaches to calculate the DT include first computing the VD, we define it here as well (see Figure 2c for an example).

DEFINITION 7 - VORONOI DIAGRAM
Let $\{v_1, v_2, ..., v_n\}$ be a set S of n vertices called sites in the plane $\mathbb{R}^2$. We define the Voronoi region for a site $v_i \in S$ as $VR_i := \{p \in \mathbb{R}^2 \mid d(p, v_i) \leq d(p, q), \forall q \in S, q \neq v_i\}$, where $d(.,.)$ denotes the Euclidean distance metric. And finally the Voronoi Diagram $VD(S) := \bigcup_i \partial VR_i$, where ∂VR_i means the boundary of VR_i (i.e., each vertex has equal distance to at least two sites).

The Voronoi diagram is named after, and was studied by, Georgy Feodosevich Voronoi in his work from 1908 [Vor09]. It is also known as Voronoi tessellation, Voronoi decomposition or Voronoi partition. In this work we will only use it for the construction of constrained triangulations, which is why we will not discuss it in more detail.

The constrained Delaunay triangulation (CDT) is a generalized Delaunay triangulation which allows required segments (constraints) as part of the triangulation.

DEFINITION 8 - [CHE89] CONSTRAINED DELAUNAY TRIANGULATION
Let G be a planar straight-line graph (PSLG). A triangulation T is a constrained Delaunay triangulation (CDT) of G if each edge of G is an edge of T and for each remaining edge e of T there exists a disc d with the following properties:

1. The endpoints of edge e are on the boundary of d.
2. If any vertex v of G is in the interior of d then it cannot be seen from at least one of the endpoints of e (i.e., if one draws the line segments from v to each endpoint of e then at least one of the line segments crosses an edge of G).

1.2 APPLICATION

Triangulation is a major topic in computational geometry. It is widely used in areas of geometric data processing. In geographic information systems (GIS) triangulation is used to represent real world data via vector graphics, e.g., TIN (triangulated irregular network). The field of robotics uses visibility graphs for motion planning. Polygon triangulation in $\mathbb{R}^2$ is one way to generate such a graph. Triangulation is also used for modeling surfaces e.g. computer aided design (CAD). Also some hidden surface removal (HSR) algorithms use triangulation to conduct visibility checks.

1.3 HISTORY

In 1975, Shamos and Hoey illustrate the Voronoi diagram in their work *Closest-point problems*, where they show an $\mathcal{O}(n \log n)$ bound for many closest-point related problems [SH75]. In 1980, a divide and conquer algorithm was published by Lee and Schachter which can compute the Delaunay triangulation in $\mathcal{O}(n \log n)$ time [LS80]. In 1985, Guibas and Stolfi propose two further algorithms, one to produce the Voronoi diagram in $\mathcal{O}(n \log n)$ time, and another to insert one site in $\mathcal{O}(n)$ time [GS85].

In 1986, Fortune publishes his work *A Sweepline Algorithm for Voronoi Diagrams* [For86], where he describes an $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space algorithm to generate a VD. The idea for the computation of a VD by the use of a sweep-line algorithm was thought to be impossible for a long time. He introduces the notion of a beach-line which is composed from parabolic arcs defined by the input vertices behind the sweep-line. Also in 1986, Lee and Lin define the constrained Delaunay triangulation in their work *Generalized Delaunay triangulation for planar graphs* [LL86].

The triangulation of a simple polygon faster than $\mathcal{O}(n \log n)$ was an unsolved problem until 1988. Tarjan and Van Wyk develop an algorithm that runs in $\mathcal{O}(n \log \log n)$ time [TW88]. In the following years several algorithms where engineered with $\mathcal{O}(n \log^* n)$ time complexity [CTW88] [Sei91] [CCT91]. In 1989, Chew discloses a divide and conquer approach to construct the constrained Delaunay triangulation in $\mathcal{O}(n \log n)$ time [Che89].

In 1991 Chazelle published his now famous paper *Triangulating a simple polygon in linear time* [Cha91]. It shows, as stated in the title, how to triangulate a simple polygon in $\mathcal{O}(n)$ time. The algorithm is very complex and too complicated to implement in practice. Chazelle also states that a test wether or not a polygon is simple can be accomplished in $\mathcal{O}(n)$ time. In the same year Seidel generalized Fortune's sweepline algorithm to compute constrained Delaunay triangulations [Sei91].

In 1998 Chin and Wang showed that a constrained Delaunay triangulation of a simple polygon can be computed in optimal $\mathcal{O}(n)$ time [CW98].

1.4 CONTRIBUTION

This work summarizes the most common triangulation algorithms. We will survey a few current publications in the direction of CDT computation as well as GPU computing approaches. Two parallel ear-clipping algorithms will be discussed in detail. Both were implemented and tested and the results are visualized.

1.5 OUTLINE

In this chapter we saw the definitions of the different triangulation types. Some applications were listed to get an idea why triangulation is an important topic. Also we saw in a short history who was involved in the research in this field.

Next we will discuss different triangulation algorithms in detail in Chapter 2. We start with the standard algorithms which are still in use and end with some parallel variants, which compute by the use of multicore architecture as well as GPU.

In Chapter 3 we will look at FIST, which is an implementation of one of those algorithms. We see details of the implementation and also specific aspects how special input is handled.

Then in Chapter 4 our two parallel extensions of FIST are discussed.

Finally, in Chapter 5 we present our experimental results comparing our parallel versions to the regular implementation.

2 ALGORITHMS

In this chapter we discuss different triangulation algorithms which were published over the last decade. We start with ear-clipping in Section 2.1, which is a simple $\mathcal{O}(n^2)$ triangulation algorithm. Then we will discuss triangulation via subdivision of the polygon into its monotone parts, which are triangulated separately. This algorithm runs in $\mathcal{O}(n \log n)$ time (see Section 2.2).

Since quality triangulations are preferred for most applications, we will examine a few constrained Delaunay triangulation algorithms as well. The first one is a sweep-line algorithm for CDT computation. It creates the CTD with a Fortune-like sweep-line approach (see Section 2.3). Then we review an algorithm to compose a CDT using an incremental construction method in Section 2.4. At last we discuss a CDT computation via GPU in Section 2.5.

2.1 EAR CLIPPING

The ear-clipping algorithm has an $\mathcal{O}(n^2)$ runtime but is easy to implement and can be very fast in practice. In Chapter 3 we will see details about our implementation of this algorithm. Ear clipping is based on Meisters two-ear theorem:

The next definition was given by Meisters [Mei75] and also applied by Held [Hel01].

DEFINITION 9 - EAR
Three consecutive vertices v_{i-1}, v_i, v_{i+1} of a simple polygon P form an ear of P if v_{i-1} and v_{i+1} constitutes a diagonal of P.

We defined the diagonal in Definition 4. For the purpose of simplicity, we will sometimes refer to an ear v_i, v_j, v_k of a polygon P, as an ear of P at v_j.

DEFINITION 10 - NON-OVERLAPPING [MEI75]
Two ears are non-overlapping if their interior regions are disjoint, otherwise they are overlapping.

THEOREM 1 - TWO-EAR-THEOREM [MEI75]
Except for triangles, every simple polygon has at least two non-overlapping ears.

DEFINITION 11 - REMOVING AN EAR
Let P be a polygon on a plane and $v_1, ..., v_n$ its n consecutive vertices. Let v_{i-1}, v_i, v_{i+1} be an ear of P. If that ear is removed then P defined by $v_1, ..., v_{i-1}, v_i, v_{i+1}, ..., v_n$ is transformed to $P' = v_1, ..., v_{i-1}, v_{i+1}, ...,$ v_n. The remaining contour consists of $n-1$ vertices.

The following proof follows the exposition given by Meisters [Mei75].

PROOF - TWO-EAR-THEOREM
Proof by induction on the number of n vertices of a simple polygon P. For $n = 4$ the polygon v_1, v_2, v_3, v_4 can have at most one reflex vertex. If so, let v_3 be reflex, the two non-overlapping ears of P are formed by v_1, v_2, v_3 and v_3, v_4, v_1, as the hypothesis states. If all vertices are convex the same two ears are still non-overlapping.

Let $n > 4$ and v be a vertex of P with an internal angle less than π. Let v_-, v, v_+ be three consecutive vertices of P.

CASE 1 v_-, v, v_+ form an ear of P (see Figure 3a). If this ear is removed from P the resulting polygon P' is either a triangle and, therefore, forms another non-overlapping ear of P. Or otherwise P' is a simple polygon with $n > 3$ and consists of one vertex less than P. The induction hypothesis states that P' has again two non-overlapping ears E_1 and E_2. Considering E_1 and E_2 are non-overlapping at least one of them is not at v_- nor at v_+, let it be E_1. Since all ears of P', except ears at v_- or v_+, are also ears of P, the two ears E_1 and v_-, v, v_+ are non-overlapping ears of P.

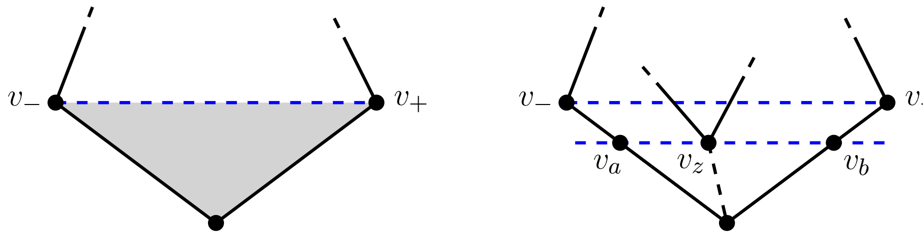
CASE 2 P has no ear at v. Then the triangle $\Delta(v_-, v, v_+)$ contains at least one vertex in its interior or on the diagonal $\overline{v_- v_+}$. From all those vertices we choose the one closest to v and denote it by v_z. Let $\overline{v_a v_b}$ be the line segment which is parallel to $\overline{v_- v_+}$ and intersects v_z. This line segment intersects the polygon at v_a and v_b (see Figure 3b).

The triangle $\Delta(v_a v v_b)$ contains no further vertex. That means the diagonal $\overline{v_z v}$ lies entirely in the interior of P and can be used to split P into two simple polygons $P_1 = v, v_z, ..., v_-$ and $P_2 = v, v_+, ..., v_z$. Either of them consist of less vertices then P since P_1 does not contain v_+ and P_2 does not contain v_-.

CASE 2A P_1 is a triangle and P_2 is not a triangle (if P_2 would be a triangle as well then P_1 and P_2 form the two non-overlapping ears). Then v, v_z, v_- form an ear of P. As the hypothesis states P_2 must contain two non-overlapping ears E_1 and E_2. One of those ears is not at v nor at v_z, lets say E_1. Due to the non-overlapping property E_1 and v, v_z, v_- form two ears of P.

CASE 2B P_1 is not a triangle. Then again the hypothesis states that P_1 and P_2 have each two non-overlapping ears. Since v and v_z are the only vertices contained in both polygons, at least one ear of P_1 is not containing them, the same holds for P_2.

$\square$



(a) Case 1 (b) Case 2

Figure 3: Polygon P: In (a) $\Delta(v, v_+, v_-)$ forms an ear of P. In (b) v_z lies in the interior of that triangle.

Naively implemented ear clipping would take $\mathcal{O}(n^3)$ time: $\mathcal{O}(n)$ ears to complete the triangulation and $\mathcal{O}(n^2)$ to find an ear. This is due to the fact that we may need to check $\mathcal{O}(n)$ vertex triples whether they form an ear and for each test we need to test $\mathcal{O}(n)$ vertices.

The basic idea of the ear-clipping algorithm is to first classify all ears and store them in some sort of queue. Then we start with the clipping process by taking out ears one by one and store them in a triangle list. For each ear v_i, v_j, v_k which we remove, we have to re-classify its two outer vertices v_i and v_k. These vertices might be ears now and have to be stored in the queue as well.

The classification step has an $\mathcal{O}(n^2)$ runtime. In total we have to check n vertex triples v_{i-1}, v_i, v_{i+1}. Then, on each triple we have to check all n vertices whether one of them lies inside the triangle $\Delta(v_{i-1}, v_i, v_{i+1})$. If the interior region of the triangle is empty and also no vertex lies on the line segment $\overline{v_{i-1}v_{i+1}}$ then $v_{i-1}v_iv_{i+1}$ form an ear.

The clipping step takes $\mathcal{O}(n^2)$ time since we need to clip n ears, and on each ear v_{i-1}, v_i, v_{i+1} we have to re-/classify v_{i-1} and v_{i+1}. This re-/classification takes $\mathcal{O}(n)$ time like above and gives us an overall $\mathcal{O}(n^2)$ runtime.

In practice various mechanisms like grids can be used to speed up the runtime drastically (see Chapter 3) but since they are heuristic in nature, they will not reduce the theoretical bound.

2.2 TRIANGULATION USING MONOTONE POLYGONS

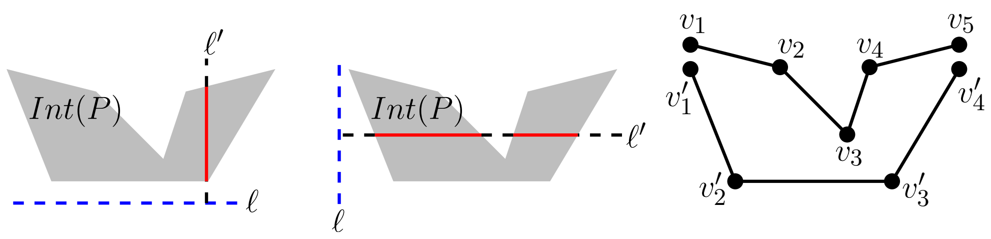
A simple $\mathcal{O}(n \log n)$ time triangulation algorithm exists which uses monotone subdivision. The idea is that a monotone polygon can be triangulated easily in $\mathcal{O}(n)$ time. To enable the triangulation of an arbitrary simple polygon, it has to be subdivided into monotone parts. This monotone subdivision can be done in $\mathcal{O}(n \log n)$ time. Since the subdivision only adds a linear amount of diagonals to the polygon, the triangulation of the monotone subpolygons still only needs $\mathcal{O}(n)$ time, so we get an overall $\mathcal{O}(n \log n)$ time algorithm.

In this section we follow the description given in the book *Computational Geometry* [BCKO08] of the algorithm originally published in the article *Triangulating a Simple Polygon* in 1978 [GJPT78].

DEFINITION 12 - MONOTONE POLYGON
A simple polygon P is monotone with respect to a line ℓ if for all lines

ℓ' orthogonal to ℓ, the intersection of $Int(P)$ with ℓ' results in at most one connected component (see Figure 4).

Another definition of a monotone polygon is that it consists of two monotone chains (see Figure 4c). In some cases we require a *strictly monotone* polygon. Strictly monotone means that on either monotone chain two vertices do not hold the same coordinates relative to the monotony axis.



(a) x-monotone (b) not y-monotone (c) two monotone chains

Figure 4: In (a) an x-monotone polygon, in (b), a not y-monotone polygon, and in (c), two monotone chains.

2.2.1 *Monotone Subdivision*

We start at the top-most vertex of our simple polygon P and walk down on either side of our contour. On a vertex v_i where we change direction and walk up again we know that the y-monotony is not given (see Figure 5).

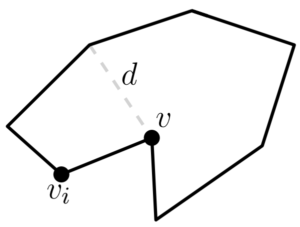


Figure 5: At the vertex v_i we can see that this polygon is not y-monotone.

When both adjacent edges of a vertex v lead downwards and the interior of the polygon P lies locally above, we should add a diagonal d going up from v. Adding such a diagonal means we subdivide P and that the vertex v will be contained in both sub-polygons. After this subdivision, one adjacent edge of v is going up and the other is going down. This takes place in either sub-polygon, thus both sub-polygons are now either y-monotone or at least the part we changed is no longer changing the direction.

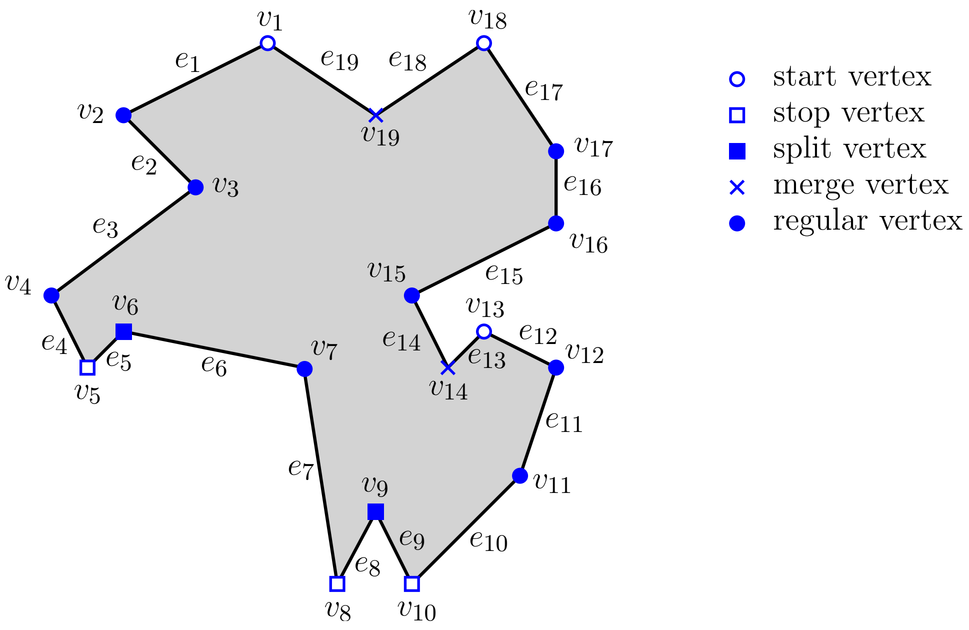
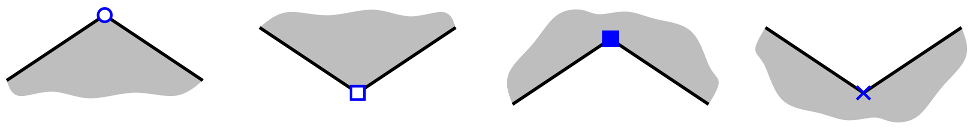


Figure 6: A simple polygon containing all vertex cases.

For this algorithm to avoid special cases with equal y-coordinates the notion of above and below is defined as follows:

DEFINITION 13 - [BCKO08]
A vertex p is *below* another vertex q means that $p_y < q_y$ or $p_y = q_y \wedge p_x > q_x$. A vertex p is *above* another vertex q means that $p_y > q_y$ or $p_y = q_y \wedge p_x < q_x$.

There are several vertex cases to differentiate: *start*, *stop*, *merge* and *split* vertex (see Figure 6). The rest of the vertices are *regular*, which means they have one adjacent edge going down and one going up.



(a) start vertex (b) stop vertex (c) split vertex (d) merge vertex

Figure 7: All four vertex cases uses for the monotone subdivision.

In a start vertex v (see Figure 7a) both adjacent neighbors lie below v and the internal angle between the adjacent edges is less than π. If that angle is greater than π, then v is a split vertex (see Figure 7c).

Next, we have the stop vertex (see Figure 7b). Here, both adjacent edges go up, meaning both neighbors lie above v and the interior angle is less than π. If the angle is greater than π, it is a merge vertex (see Figure 7d); the interior of P lies locally below it.

The correctness of this algorithm depends on the following lemma:

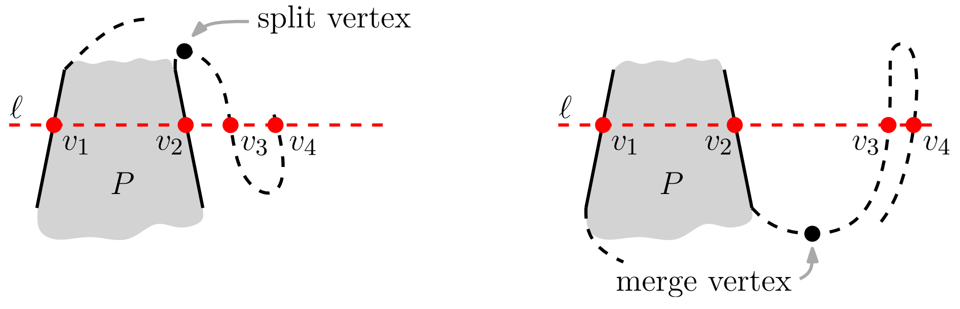
LEMMA 5 A simple polygon is monotone relative to the y-axis if it does not contain any merge or split vertices.

The following proof follows the exposition given by de Berg et al. [BCKO08].

PROOF
Let P be a simple polygon that is not y-monotone. Since P is not y-monotone there exists by definition a line orthogonal to the y-axis that intersects $Int(P)$ and creates more than one connected component. Let ℓ be such a line. It follows that ℓ intersects the contour of P at least four times. We sort those vertices by their x-coordinate an denote them $v_1, ..., v_n$. Then we walk along the contour from v_2 to v_3. On the way we have to find a maximum, which means a split vertex, or minimum, which would be a merge vertex (see Figure 8).

□

The idea of the algorithm is to walk through the contour with a line sweep from top to bottom and create diagonals along the way from split vertex to merge vertex. This divides the polygon into sub-polygons which contain neither split nor merge vertex and, therefore, are monotone due to Lemma 5.



(a) first case (b) second case

Figure 8: Two illustrations to support the proof above.

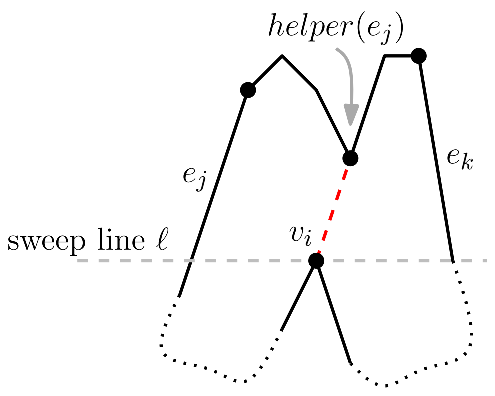
Let P be a simple polygon and the vertices counterclockwise (CCW) along the contour be denoted by $v_1, ..., v_n$. Let edges $e_1 = \overline{v_1 v_2}, ...,$ $e_{n-1} = \overline{v_{n-1} v_n}$ and of course $e_n = \overline{v_n v_1}$ (see Figure 6). First, we sort the vertices of P by their y-coordinate and store them in a queue Q. If equal y-coordinates emerge then the left vertex is taken first, according to our previous definition of above and below (see Definition 13).

If we encounter a split vertex v_i, we have to add a diagonal upwards. So how does one find a diagonal which lies entirely inside of the polygon P? The sweep line always saves the next left and right edge e_j, e_k from our current vertex v_i. Then we connect v_i to the lowest vertex between e_j and e_k. If there is no such vertex, we connect v_i to the lowest top vertex of the edges e_j and e_k (see Figure 9a).

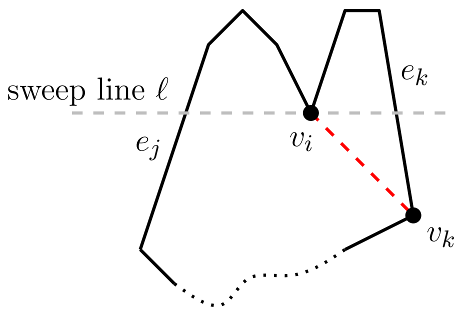
DEFINITION 14 - [BCKO08]
Let $helper(e_j)$ be the lowest vertex v_x above the sweep line ℓ such that the horizontal line segment which starts at v_x and intersects e_j before any other edge on the side of e_j.

Each edge maintains its own $helper()$ which can change during the sweep.

When we encounter a merge vertex v_i, we need to add a diagonal down, which seems more complex than before since we only have computed the subdivision until the split line. The idea is to connect the merge vertex v_i to the highest vertex below the sweep line. We do not know that vertex right now, but when we reach v_k and want to replace $helper(e_j)$ with v_k we can see that the old $helper(e_j)$ is a merge vertex and add the diagonal $\overline{v_i v_k}$. This means, every time we reset a $helper()$ we check if the current $helper$ vertex is a merge vertex. If so, we first add the diagonal and then reset that $helper$. If we have to handle a split vertex and $helper()$ refers to a merge vertex (like in Figure 9a), we would take care of both at the same time.



(a) Example for handling a split vertex v_i.



(b) Example for handling a merge vertex v_i.

Figure 9: Examples for both split and merge vertices.

Important as well is how we find the next left or right edge from each vertex. For this we save the *sweep line status* in a data structure containing the edges currently intersecting the sweep line in a left-to-right order. At each vertex this sweep line status is updated. Only edges which have the interior of the polygon on the right are stored.

Now we review the algorithm to verify the overall runtime. At the start we have to sort the vertices according to their y-coordinates and store them in a queue Q, which is done in $\mathcal{O}(n \log n)$ time. Next, we start the line sweep which has to dequeue each of the n vertices. Since Q is already sorted, it takes $\mathcal{O}(1)$ to get a vertex and then $\mathcal{O}(\log n)$ time to process it. The processing involves to find the next left edge, which is inside the data structure containing the sweep line status. As mentioned above this data structure is already in a left to right order, thus the search can be conducted in $\mathcal{O}(\log n)$ time. If a new edge is to be inserted into the sweep line status, it takes again $\mathcal{O}(\log n)$ time, but on each vertex at most one edge has to be removed or inserted. At last, the insertion of a diagonal takes $\mathcal{O}(1)$ time. This gives us an overall $\mathcal{O}(n \log n)$ runtime for the subdivision.

2.2.2 *Triangulation of a Monotone Polygon*

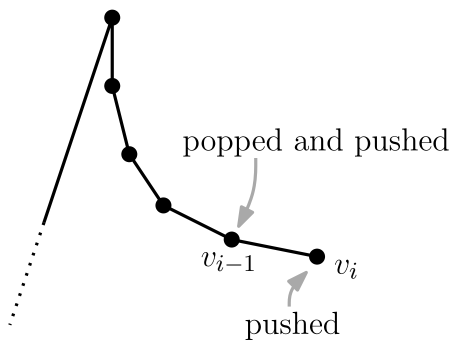
Let P be a y-monotone polygon with n vertices. To avoid special cases we let P be strictly y-monotone. That means, if we split up P into its two monotone chains (left and right), we can walk down along either contour, by starting at the top and always go downwards (decreasing y-coordinate). This makes the triangulation very easy. The idea is to insert diagonals along the way whenever possible.

We assume we have all n vertices of our polygon P in a doubly-linked list L. We also use a stack S as an auxiliary data structure. The algorithm handles the vertices by decreasing y-coordinate, starting at the top (start) vertex v_1. While processing a vertex v_i, there remain only unfinished vertices (those which still need some diagonals) on the contour above. The stack S is used to store those vertices in the following order. The last processed vertex v_{i-1} will be on top of S, and the vertex with the highest y value on the bottom. After we finished with vertex v_i, we push it on the stack as well, since it is not finished.

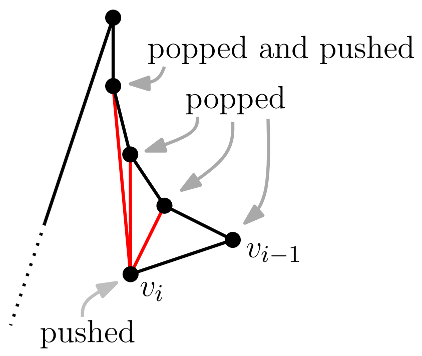
While processing a vertex v_i we distinguish two cases: First case is that the previous vertex v_{i-1} is on the same contour chain as v_i (see Figure 10), the second case is that v_{i-1} is on the opposite side (see Figure 11).

In the first case (see Figure 10), where v_i is on the same side as the vertex v_{i-1}, we get some kind of reflex curve along the points remaining on the stack S. If v_{i-1} is also reflex, we just get an even longer reflex curves by pushing v_i onto S. If v_{i-1} is convex we can add a diagonal $\overline{v_{i-2}v_i}$.

This algorithm just pops the last two vertices v_{i-1}, v_{i-2}. If the internal angle of v_{i-1} is convex, i.e., less than π, we insert the diagonal $\overline{v_{i-2}v_i}$, store the triangle $\Delta(v_{i-2}, v_{i-1}, v_i)$, and remove the convex vertex v_{i-1} from the contour loop. Now the next vertex v_{i-3} is popped from S and this process is repeated until an angle is reflex. Then



(a) v_{i-1} is reflex



(b) v_{i-1} is convex

Figure 10: Monotone polygon: first case.

no further diagonal can be added and the last vertex taken from the stack and v_i are pushed onto S.

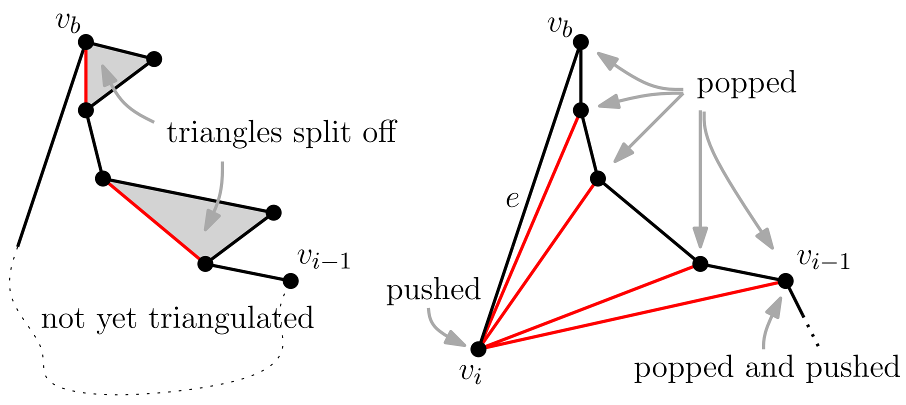


Figure 11: Monotone polygon: second case.

In the second case (see Figure 11), the processed vertex v_i is on the opposite side of v_{i-1}. This means that all vertices on the stack are from the opposite chain, except the bottom vertex v_b. This means we can add diagonals to all of them and remove all vertices from the stack. No diagonal is needed to the bottom vertex v_b, since the contour edge e, from v_b to v_i, already exists. At last we need to push v_{i-1} and then v_i back on the stack, since they are not finished.

The algorithm has the following runtime: The initialization is done in $\mathcal{O}(1)$. The main process to walk through the monotone chains vertex by vertex takes $\mathcal{O}(n)$. It is possible that the effort to process one vertex takes linear time, but at each step at most two vertices are pushed onto the stack. Since the number of pops can not exceed the number of pushes, this whole process runs in $\mathcal{O}(n)$ as well as the final step. This means an y-monotone polygon can be triangulated in linear time.

What we want is to combine the monotone subdivision, which runs in $\mathcal{O}(n \log n)$ time, and the triangulation of a monotone polygon, which runs, as shown above, in $\mathcal{O}(n)$ time. Due to the fact that, while splitting into monotone sub-polygons, we only add a linear amount of diagonals, we also add only a linear amount of vertices. This means that triangulation of those sub-polygons still runs in $\mathcal{O}(n)$ time, resulting in an overall $\mathcal{O}(n \log n)$ time algorithm to triangulate a simple polygon.

2.3 A SWEEP-LINE ALGORITHM FOR CONSTRAINED DELAUNAY TRIANGULATION

We defined the constrained Delaunay triangulation (CDT) in Chapter 1, Definition 8. In this section, we will discuss the algorithm published by Domiter and Žalik in 2008 [Dv08].

This algorithm uses a Fortune-like sweep-line approach to generate the CDT directly. Below, in Section 2.5, we will see another approach which generates the Voronoi diagram first, continues with the Delaunay triangulation and finally, by adding the constraints, generates the CDT.

In a sweep-line algorithm, as the sweep-line travels from $-\infty$ to $+\infty$, the computation below the sweep-line is always complete. Usually, no guarantee is stated for the state above it.

The counterpart to Fortune's beach-line is called *advancing front*. It consists of the topmost edges below the sweep-line (see the blue dashed line segments in Figure 12). Necessary data structures are a doubly-linked list L to store the advancing front (AF) and a queue Q for the input vertices. A vertex data field v_i stores its coordinates as well as references about existing adjacent edges and their state (starting / ending).

The algorithm is split into three parts: initialization, sweeping and finalization. They will be discussed in detail in the following sections.

2.3.1 *Initialization*

First, the input vertices $v_1, v_2, ..., v_n$ are sorted by their y-coordinates. Then, two artificial vertices v_{-1}, v_0 are added to avoid the occurrence of special cases. In the finalization step (Section 2.3.3), the artificial vertices will be removed.

$$v_{-1} = (x_{min} - \delta_x, y_{min} - \delta_y)$$
$$v_0 = (x_{max} + \delta_x, y_{min} - \delta_y)$$

where $\delta_x = \alpha * (x_{max} - x_{min})$, $\delta_y = \alpha * (y_{max} - y_{min})$ and $\alpha > 0$. Special cases occur when not only the advancing front but also the triangulation below it has to be changed. Adding v_{-1} and v_0 avoids that problem completely.

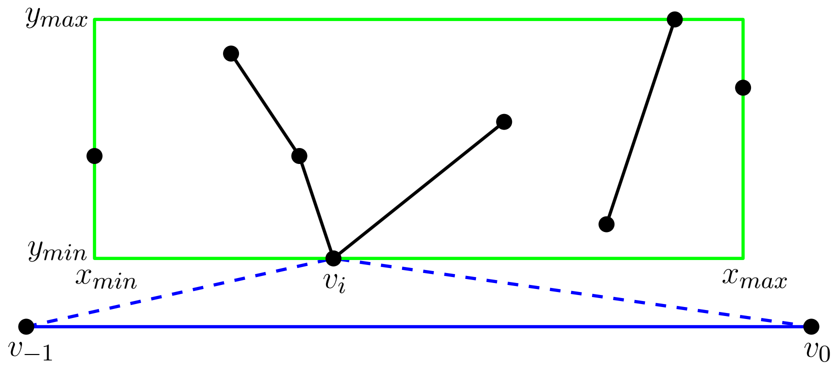


Figure 12: Initialization: the green bounding box to set v_{-1} and v_0, the first triangle $\Delta(v_{-1}, v_0, v_i)$ and the AF v_{-1},v_i,v_0 in blue dashed line.

In the discussed article, α is a constant and set to 0.3. The vertices v_{-1} and v_0 are used to create the first triangle $\Delta(v_{-1}, v_i, v_1)$. In this triangle v_i is the first vertex from the queue and has the lowest y-

coordinate. This will initialize the advancing front (AF) by adding v_{-1}, v_i, v_0 to the list L.

See Figure 12 for an example of the previously explained initialization.

2.3.2 *Sweeping*

During the sweep phase, two types of events can be distinguished from each other: point events and edge events.

A point event denotes the insertion of a vertex into the triangulation. Two cases are to be differentiated when inserting a vertex v. The first one is the *middle case*. It occurs when v lies strictly between two vertices of the advancing front (AF). The second one is the *left case* which is at hand when v lies exactly above an AF-vertex.

An edge event inserts a vertex and the adjacent edges (constraints). Every constraint is always associated to its top vertex (the vertex with the higher y-coordinate). When processing a vertex v_i which references to an edge where v_i is not its upper end point, this edge is not taken into account. The end point of an edge denotes its upper vertex whereas the start point denotes its lower vertex.

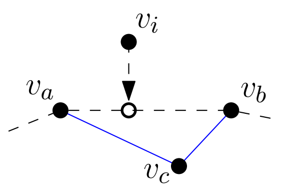
Point Event

When a point event occurs, the appropriate spot of insertion has to be found in the AF. The geometric search is accelerated by a hash table. At this point, we can distinguish between the middle case and the left case which were mentioned in the section above.

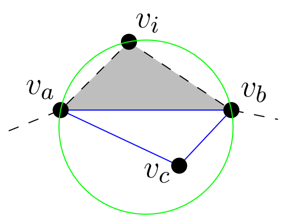
MIDDLE CASE A vertex v_i has to be inserted and is projected vertically on the AF. This projected vertex lies strictly between two consecutive AF-vertices v_a and v_b. In this case, the triangle $\Delta(v_a, v_b, v_i)$ is formed and v_i is inserted between v_a and v_b into the AF (see Figure 13a).

LEFT CASE Like in the middle case, a vertex v_i has to be inserted. Now, its projection on the AF coincides exactly with an AF-vertex v_b. In this case, two triangles $\Delta(v_a, v_b, v_i)$ and $\Delta(v_i, v_b, v_c)$

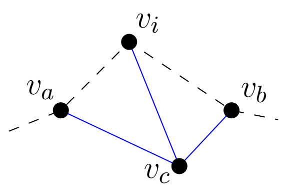
are added where v_a, v_b, v_c are consecutive vertices of the AF. At last, v_b is replaced by v_i in the AF (see Figure 13d).



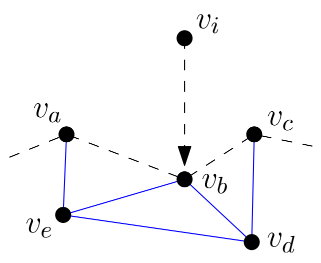
(a) Vertex v_i is projected on the AF.



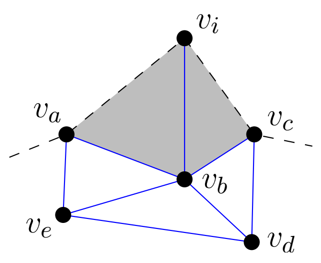
(b) AF updated, triangle $\Delta(v_a v_b v_i)$ added.



(c) Legalization of edge $\overline{v_i v_c}$.



(d) v_i is projected on the AF. Projection coincides with v_b.



(e) Two triangles are added: $\Delta(v_a v_b v_i)$ and $\Delta(v_i v_b v_c)$.

Figure 13: Sweep-line CDT: (a, b, c) middle and (d, e) left case.

After one of the described cases is handled, a legalization process is carried out. This is a mechanism which was introduced by Lawson in his work *Software for C1 surface interpolation* in 1977 [Law77]. He shows that every triangulation can be transformed into a DT by applying the empty-circle test on each edge which is shared by two triangles.

A small example of this legalization process is pictured in Figure 14.

Additionally, this algorithm creates triangles to visible points on the AF. This can lead to an unbalanced triangulation. Therefore two heuristics are introduced to reduce the workload for the legalization process.

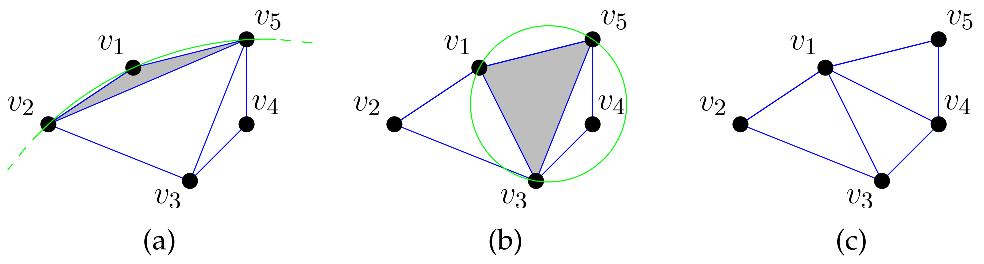


Figure 14: Sweep-line CDT: legalization. In (a), the in-circle test is performed on $\Delta(v_2v_5v_6)$. Since the circle is not empty, an edge flip is performed on $\overline{v_2v_5}$. In (b), another in-circle test is applied to $\Delta(v_3v_5v_6)$ and another edge flip to $\overline{v_3v_5}$, which can be seen in the resulting DT in (c).

1. If the angle between the newly inserted triangle and the AF is smaller than $\pi/2$, triangles are added until this property no longer holds (see Figure 15).

2. To avoid the appearance of basins, the fluctuation of the AF has to be controlled. A basin is like a sink along the AF where no triangles were generated yet. Therefore, if the angle between AF-segments is larger than $3\pi/4$, the basin is filled with triangles (see Figure 16). The notion basin was also defined by Žalik in 2005 [Ž05].

Edge Event

An edge event takes place whenever an edge end-point (upper vertex) is reached. As explained in Section 2.3.2, each vertex references its adjacent edges.

First, the current vertex v is inserted like in the subsection *Point Event*. The insertion of its constraint c (edge) starts with the search for the first intersected triangle. The first triangle can be found by evaluating the direction vector of c and comparing it to the ones, from the triangles which contain v. Every further triangle is found by traversing through the triangulation, using adjacency links. This process is explained in full detail in the work of Anglada et al. from 1997 [Ang97].

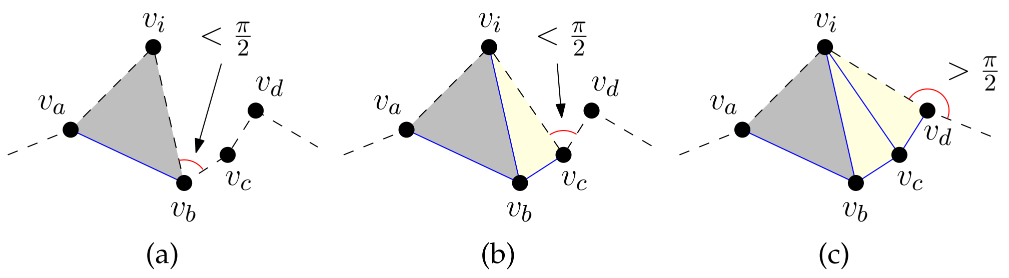


Figure 15: Heuristic 1: The newly inserted triangle $\Delta(v_a v_i v_b)$ changes the AF. The angle between the AF-segments $\overline{v_i v_b}$ and $\overline{v_b v_c}$ is $< \pi/2$. In (b), the triangle $\Delta(v_i v_b v_c)$ is added, and the AF is updated. Again, the angle between the AF-segments is $< \pi/2$ and in (c) the triangle $\Delta(v_i v_c v_d)$ is inserted. The angle between the AF-segments is now $> \pi/2$ and the inserting process stops.

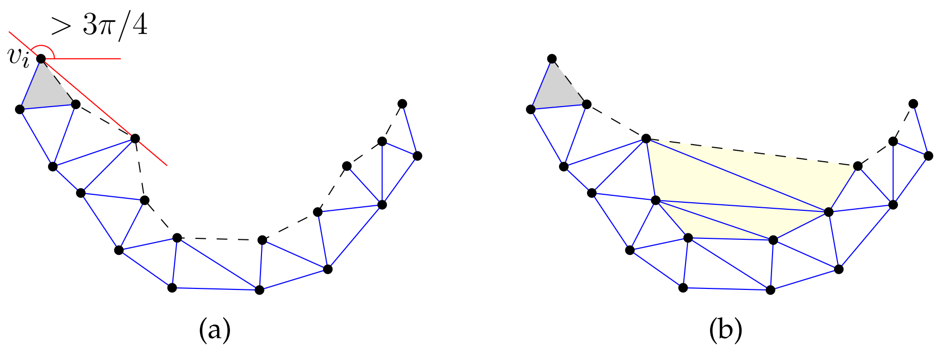


Figure 16: Heuristic 2: In (a), v_i is inserted and due to the second heuristic a basin is detected, and in (b), that basin is filled with triangles.

The idea is to determine the position of a triangle relative to c. After all intersecting triangles are found, they are removed from the triangulation. Their vertices are stored in Π_u and Π_l, where Π_u stores the vertices above the constraint and Π_l those below it (see an example for this process in Figure 17a-c).

Two sub-polygons are formed by using the vertices stored in Π_u and Π_l. Then those sub-polygons are re-triangulated. The following cases may occur while inserting a constraint c:

- If c coincides with an edge e of a triangle, e is marked as fixed and must not be changed after this step.

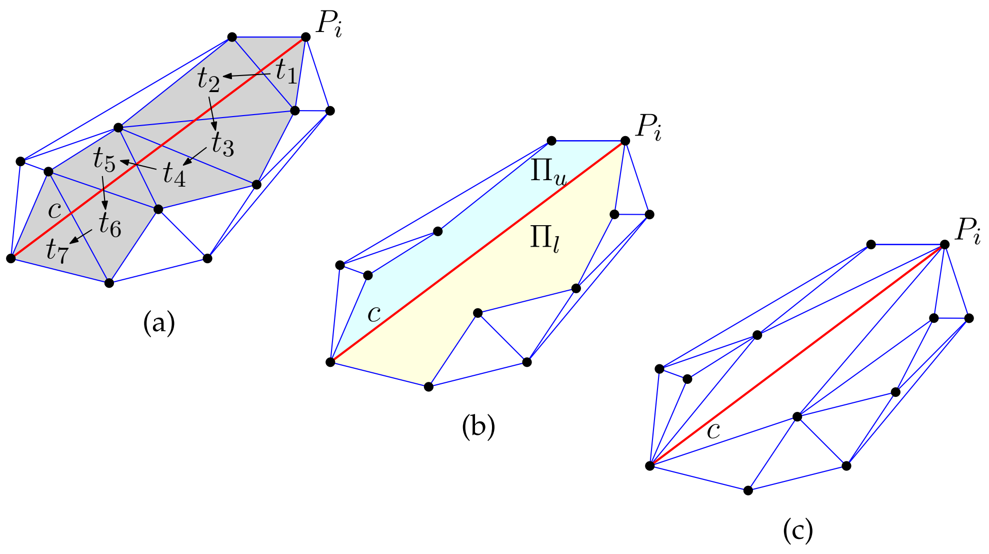


Figure 17: Sweep-line CDT triangle traversal: In (a), the triangles $t_1, ..., t_7$ intersecting the constraint c are found. In (b), two sub-polygons are created by using Π_u and Π_l. In (c), these sub-polygons are triangulated separately without intersecting c.

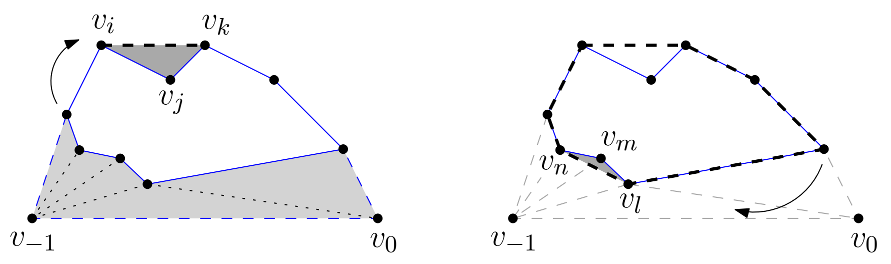
- If c is entirely above the AF, no triangle is pierced. In that case Π_u is empty and Π_l is created by an *AF-traversal*. In the AF-traversal process we walk through the AF one vertex at a time and insert it in Π_l if it is still below c. Then Π_l is triangulated.
- If c is partially above and partially below the AF: as long as c is above the AF, an AF-traversal is conduced. Where c is intersecting with the AF the algorithm switches to triangle traversal (see explanation of the triangle traversal process in full detail in [Ang97]).

2.3.3 *Finalization*

Two steps remain to complete this triangulation algorithm. First, the CDT should have the convex hull as its border. Second, the two artificial vertices v_{-1} and v_0 and all triangles containing at least one of them have to be removed.

The upper convex hull is created by walking through the AF from left to right. The algorithm starts at the beginning, which is v_{-1}. It takes vertex-triples and calculates their signed area. If positive, the triangle defined by that vertex-triple is added and legalized. When the end of the AF is reached, which is the second artificial vertex v_0, the upper part of the convex hull is completed (see Figure 18a).

The lower convex hull is created by using the triangles containing at least one artificial vertex starting at v_0. These triangles form the lower contour C_l between v_0 on the right side and v_{-1} on the left. We walk through C_l and again use vertex-triples to evaluate if this part of the convex hull is already correct. If the signed area is negative, since C_l is traversed from right to left, a triangle has to be added and legalized. In this part the triangles containing v_{-1} or v_0 are also removed (see Figure 18b).



(a) The upper convex hull is created starting at v_{-1}.

(b) Computation of the lower convex hull starting at v_0.

Figure 18: Sweep-line CDT finalization: in (a) the upper convex hull is created by walking through the AF and adding triangle $\Delta(v_i, v_j, v_k)$, in (b) the lower convex hull is computed by removing the gray dashed triangles and adding triangle $\Delta(v_l, v_m, v_n)$.

Domiter and Žalik do not provide a runtime complexity. Yet they present runtime results which show a comparison of their algorithm with three others provided through Shewchuk's Triangle package [She96]. The benchmarking results show, that their algorithm allows a speedup of about 2 for an input of 5 million vertices and 7 million edges.

2.4 INCREMENTAL CONSTRUCTION OF CONSTRAINED DELAUNAY TRIANGULATIONS

In this section we will discuss the algorithm presented by Shewchuk and Brown in 2013 [SB13]. This algorithm computes the constrained Delaunay triangulation by incremental insertion.

The worst-case runtime complexity is $\Theta(kn^2)$, where n is the number of input vertices and k is the number of input segments. Further results were published which state that this randomized approach has a expected $\mathcal{O}(n \log n + n \log^2 k)$ runtime.

Paul Chew proposed an algorithm to construct a Voronoi diagram for a convex polygon in linear expected time in 1990 [Che90].

In the discussed article, a variation of Chew's algorithm is used for inserting vertices as well as segments. In the following paragraph, Chew's algorithm, yielding a Delaunay triangulation, is described in further detail.

Let L be the listing of the consecutive vertices of a polygon P. Let R be a listing containing a randomized permutation of L. Chew's algorithm works as follows: In the first step the vertices from R are removed one by one from the contour, until only three remain. When a vertex v is removed its adjacent vertices u and w are stored as well. The last three vertices form the first triangle of the DT (see Figure 19a-d). In the second step the removed vertices are inserted in reverse order. This means the last removed vertex is inserted first. When inserting a vertex v its adjacent vertices u and w are already part of the DT. The empty circle property of the circle defined by u, v, w is verified. If it fails all triangles containing the edge $\overline{uw}$ are removed. Then the union of the removed triangles and $\Delta(u, v, w)$ is re-triangulated by inserting edges starting at v (see Figure 19e-g). This part is also known as *Bowyer-Watson* algorithm, introduced by Bowyer and Watson in 1981. All details about that algorithm can be found in their work [Bow81] [Wat81].

Chew's algorithm runs in $\mathcal{O}(n)$ expected time for a convex polygon with n vertices. A proof for this runtime can be found in Shewchuk and Brown's work [SB13]. Essentially, the point location step is done in $\mathcal{O}(n)$ time. In average, the triangle deletion is only deleting less than four edges per inserted vertex. This leads to a expected linear runtime using Seidel's backward analysis technique [Sei92].

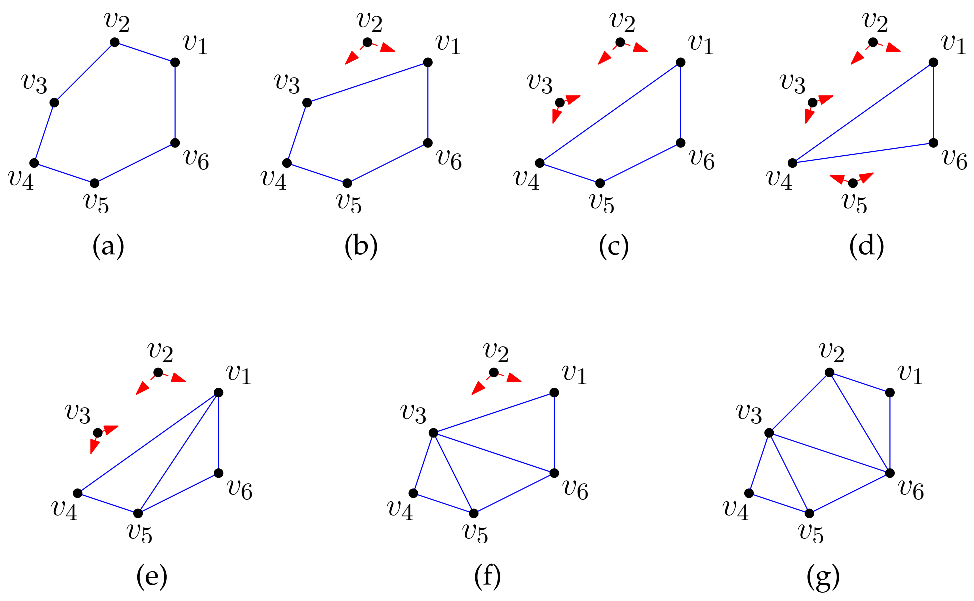


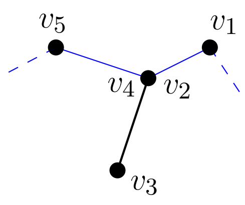
Figure 19: Incremental CDT of a Convex Poylgon: In (a), we see the input polygon. A random order $V = v_2, v_3, v_5, ...$ of the consecutive vertices is generated. The vertices of V are removed from the contour and their adjacent edges stored (b-d). In (d), only three vertices remain, which already form a triangle $\Delta(v_1 v_4 v_6)$. In (e), the insertion process starts, inserting the vertices of V in reverse order, by starting with v_5. In (g) the insertion process is finished yielding a CDT from the convex input polygon.

The algorithm presented by Shewchuk and Brown deviates from Chew's algorithm since it can handle non-convex polygons. Furthermore, the discussed algorithm is using Chew's algorithm as part of their re-triangulation process. The main differences are addressed next:

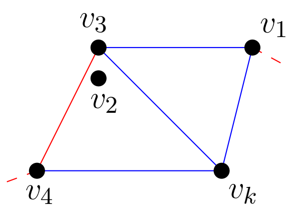
- Vertices which define constraints are inserted at the beginning.
- Constraints may have one adjacent vertex dangling inside of the polygon P. This would lead to a non-simple contour listing, since vertices would be used more than once. Those vertices are

simply inserted twice, this workaround also helps to maintain a simple list structure of the contour (see Figure 20a).

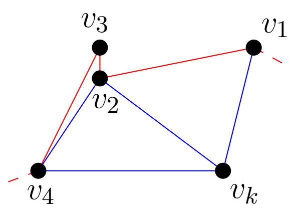
- After each inserted vertex the algorithm has obtained a CDT of the given input vertices up to this point. If a vertex v is to be inserted, the algorithm not only checks the empty-circle property of v and its adjacent vertices, but also their orientation (see Figure 20b and Figure 20c). This is essential to regain the correct contour, even when dealing with reflex vertices.



(a) v_2 is inserted a second time, as v_4.



(b) The polygon's contour is $v_1, v_2, v_3, v_4, \ldots$



(c) To insert v_2, triangles have to be removed.

Figure 20: Incremental CDT: Cases occurring in Shewchuk and Brown's algorithm.

Shewchuk and Brown's algorithm was tested against an implementation of a gift-wrapping algorithm published by Anglada [Ang97]. Details about this gift-wrapping algorithm can be found in his work from 1997 [Ang97]. The benchmarking result shows that the discussed algorithm outperforms the gift-wrapping implementation. As soon as the number of input vertices exceed $30 - 85$, depending on the input, Shewchuk and Brown's algorithm gets faster rapidly.

2.5 CONSTRAINED DELAUNAY TRIANGULATION USING GPU

Various algorithms are known to compute the constrained Delaunay triangulation (CDT) of a polygon by the use of the GPU. Some variants use a hybrid approach which means they to do parts of the computation on the CPU.

A hybrid approach to the computation of the Delaunay triangulation (DT), where the major part of the computation takes place on the GPU, was published by Rong et al. in 2008 [RTCS08].

Another interesting algorithm to compute the VD on the GPU by using a sweepcircle approach was published by Xin et al. in 2013 [XWX+13]. Fortune's sweepline algorithm performs poorly when applied in parallel, since it has to compute an overhead of approximately 90% per cell. The parallelization is done by splitting the plane into cells, applying the algorithm on multiple cells at a time. The idea of the sweepcircle is that inside the sweepcircle the Voronoi diagram is computed correctly. With this property the calculated overhead per cell is minimized and the performance for GPU computation optimized at the same time.

Since our goal is to compute the triangulation of polygons, we need a CDT. Therefore, we will discuss the publication of Qi et al. [QCT12]. Like in other CDT approaches, the first step is to compute a Voronoi diagram. Out of the VD a DT can be computed. In the end, the constraints are added and result in a CDT.

The algorithm is structured into different phases. We will discuss each phase separately in order to show the concept and the procedure.

2.5.1 *Digital Voronoi Diagram Construction*

In this phase the bounding box of the input vertices is mapped into a texture of size $m \times m$ (see Figure 21b). This texture is a binary image where each pixel is filled if at least one vertex of the input data lies in it. If more than one vertex falls into such a pixel, those vertices are removed and saved as a missing vertex for a later phase. If a vertex lies exactly between two pixels, the left pixel is chosen (see Figure 21c). Those filled pixels are the seeds to create the discrete or digital VD.

The next step is to calculate a digital Voronoi diagram (see Figure 21d; we also added a normal VD in blue dashed lines). Differing from a discrete VD, the digital Voronoi diagram does not guarantee the correctness of its dual graph to be a Delaunay triangulation. The discrete VD can be calculated using the standard flooding algorithm.

The digital VD is computed by the use of the parallel banding algorithm (PBA), which is also explained in detail in the work of Cao et al. [CTMT10]. Its idea is to create a distance map from the binary image. This map contains the minimum Euclidean distance to the next filled pixel in each cell. The cells that we calculate the distance from are the seeds from the digital VD. The value for one specific cell is calculated by using only its eight neighbors. In the discussed article the proof of correctness of an exact Euclidean distance map is provided.

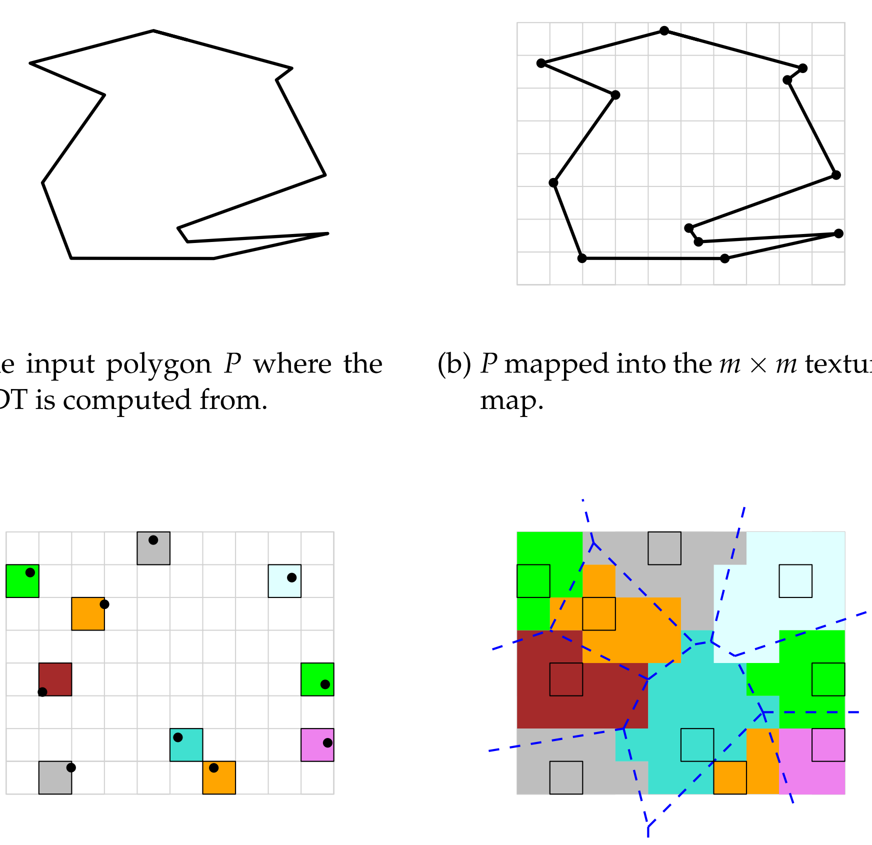
The advantage of PBA lies in running entirely in parallel while using the GPU. Its drawback is the production of so called debris. This means that the Voronoi regions can be disconnected and lead to an incorrect Delaunay triangulation. That can be avoided by a specific repair step which examines if neighboring pixels have the same color. This step runs in parallel as well and is described in detail in the discussed article by Qi et al. [QCT12].

2.5.2 *Triangulation Construction*

In the second phase, the triangles are computed by the use of the digital Voronoi diagram (DVD). The Voronoi vertices of the DVD have to be identified. They are the corners between those pixels which have at least 3 different colored pixels as neighbors (see black squares in Figure 22a). The triangles are created by walking around each Voronoi vertex. If three different colors join at such a Voronoi vertex, one triangle is added. If there are four different colors, two triangles are added into the triangulation. The seeds of the three, or at most four, adjacent Voronoi regions are used to create the vertices for the triangles (see Figure 22b).

This process can be done on the GPU by processing one texture row per thread. First, the triangles have to be counted. The offset required by each row in the data structure is calculated by using a parallel prefix sum. The triangles are then generated in parallel. This parallel prefix sum primitive is provided natively by CUDA.

CUDA (Compute Unified Device Architecture) is a *C*-like language developed by NVIDIA. It enables the use of the GPU (graphics processing unit) for parallel computation.



(a) The input polygon P where the CDT is computed from.

(b) P mapped into the $m \times m$ texture map.

(c) Texture cells and their associated vertices are marked.

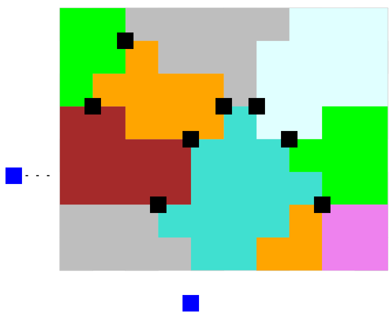
(d) PBA used to compute the digital VD of P.

Figure 21: CDT computation on the GPU: first phase.

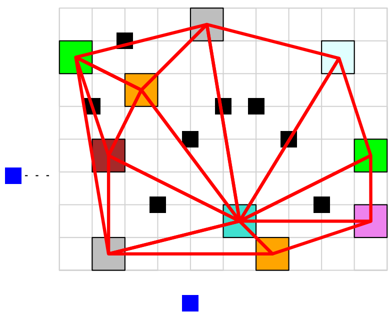
Another process, which is running concurrently on the CPU, is adding the triangles which depend on Voronoi vertices lying outside of the texture map (see blue squares in Figure 22b). This is done by using a Graham scan, named after and described by Graham in 1972 [Gra72]. Those triangles are added at the end of the triangle data structure.

2.5.3 *Shifting*

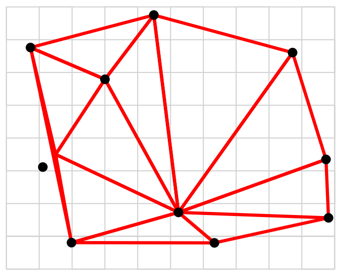
In the third phase, the vertices holding the triangulation (seeds of the DVD) are transformed back to the position of their corresponding original input vertices (see in Figure 22c and Figure 22d).



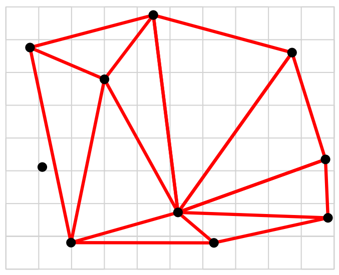
(a) Finding the digital Voronoi vertices in the DVD.



(b) Create triangles around DVV (digital Voronoi vertices).



(c) Shifting the good cases back to their original position.



(d) Removing bad case related triangles. Retriangulate hole.

Figure 22: CDT computation on the GPU: second and third phase.

There are two possible cases: the *good case*, where all neighbor triangles remain on the same side (Figure 22c), and the *bad case*, which implies that a vertex is crossing over a triangle edge (Figure 22d). Due to a good resolution in the texture map, the short shifting distance should ensure a minority of bad cases.

This process should be carried out in parallel as well. To accomplish that without corrupting the result, no two adjacent vertices are allowed to be shifted at the same time. An algorithm is checking if a good case is at hand and the vertex can be shifted, or if it has to be marked as a bad case.

This algorithm is explained in more detail in the referenced article [QCT12] by Qi et al. The testing for a bad case is done by the use of Shewchuk's orientation-test. The procedure is described in his work from 1996 [She96] where he also explains his triangulation algorithm Triangle.

After every good case is shifted, the remaining bad cases are removed from the triangulation and, like in the first phase, stored as missing vertices. When a vertex from the triangulation is removed, all adjacent edges have to be removed as well and leave a hole. This hole has to be triangulated again. Due to the star-shaped property of such a hole, it can be triangulated in linear time, for example by using Woo and Shin's algorithm [WS85]. The number of new triangles is at least one less than the triangles deleted. That ensures memory safety in parallel computation, since the slots in the triangle data structure left by the deleted triangles can be reused to store the new ones.

2.5.4 *Missing Points Insertion*

As the name suggests, the vertices which have been marked as missing are added in phase one as well as the vertices which have been removed as bad cases in the last phase (see Figure 24a).

If a vertex is inserted inside a triangle, it splits that triangle into three triangles. Adding a vertex on an edge which is shared by two triangles results in four triangles.

To ensure a good parallel search performance, the search for the triangle containing a vertex is done as follows: If we re-insert a vertex which has been removed in phase one, the search starts with the vertex v_i associated with that pixel. Since v_i is in the triangulation, only triangles containing v_i have to be tested.

If we re-insert a vertex v_j which has been removed in phase three, we test the triangles containing the vertices which shared an edge with v_j before it had been removed.

Since this is done in parallel, we have to assure that we do not insert two vertices which manipulate the same triangle. The algorithm for the point insertion is explained in full detail in the article of Qi et al. [QCT12]. It uses atomics and passes over the triangulation several times until all missing points are inserted.

2.5.5 *Adding Constraints*

Adding the constraints is carried out between phase four and phase five. Simply giving one constraint to each thread most likely results in a poor performance: One constraint could affect $\mathcal{O}(n)$ triangles while others may affect none. An even worse case would be if different constraints intersect with the same triangle.

Also in this approach, a mark phase is used. First, all triangles that intersect one constraint are found and then marked in parallel by the use of atomics. Then follows a flip phase where intersecting triangles are classified and flipped according to their case. There are four possible intersection cases: zero, one, double and concave (see Figure 23).

To solve all cases for a triangle A intersecting a constraint c, the *one-step look-ahead* method is described. It uses three triangles: The triangle intersecting c before A, and the triangle after A and of course A itself. This method converts the case at hand to a less complex case with each run. E.g. a double intersection is converted into a single intersection. This means that several runs may be required to completely solve the intersection for one constraint.

This process has to be repeated for each constraint. This algorithm and proof of its correctness are discussed in full detail in [QCT12].

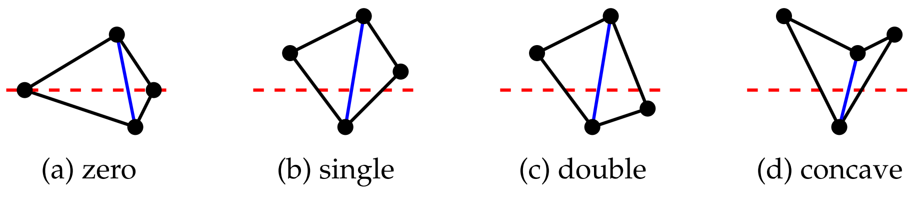
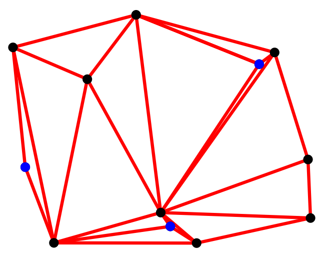


Figure 23: The four cases how a constraint (red dashed), can be intersected by an edge (blue).

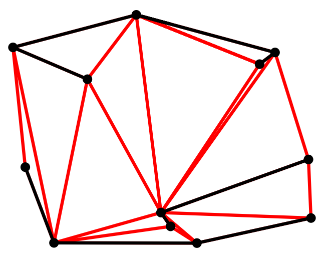
2.5.6 *Edge Flipping*

In phase five, the Delaunay property is constructed, unless there is a constraint which prevents that. For an edge $\overline{ab}$ from the triangle $\Delta(abc)$, by checking the second adjacent triangle $\Delta(abd)$, the in-circle test is conducted. If d lies inside the in-circle, the edge flip is carried out (see Figure 24b).

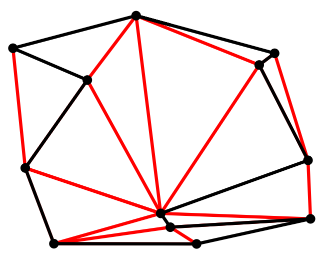
This process is executed in parallel. Again, a mark phase is followed by a flip phase. In order to mark all triangles which will be modified in the second phase, atomics are used to assure that there is never more than one thread which operates on the same triangle.



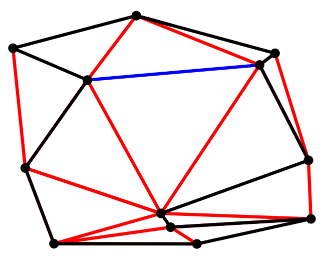
(a) Insert the missing vertices (blue) and the associated triangles.



(b) Adding constraints without flipping edges.



(c) Adding constraints, flip edges until no intersection remain.



(d) Flip unconstrained edges until inscribed circle property holds.

Figure 24: CDT computation on the GPU: fourth and fifth phase, and constraint integration.

3 FIST

FIST (fast industrial strength triangulation [Hel01]) is an ANSI C implementation of the ear clipping algorithm (described in detail in Section 2.1). The goal is to provide a fast and very robust software that can always provide a viable triangulation output. If the input data gets more and more corrupt, it should not crash but produce a triangulation which should still be useful in some way. Of course, only to a certain degree of degenerated input.

Naturally the runtime complexity in theory is $\mathcal{O}(n^2)$ but geometric hashing is used which improves the performance drastically in most practical cases.

For FIST, two sets of conditions called CE1 and CE2 are proposed. They are applied to determine whether an ear is given at a certain vertex or not. We employ this conditions defined by Held [Hel01].

LEMMA 6 - CE1
Three consecutive vertices v_{i-1}, v_i, v_{i+1} of P form an ear of P iff

1. v_i is convex,

2. the diagonal $\overline{v_{i-1}v_{i+1}}$ does not intersect any edge of P except at v_{i-1} and v_{i+1},

3. $v_{i-1} \in C(v_i, v_{i+1}, v_{i+2})$ and $v_{i+1} \in C(v_{i-2}, v_{i-1}, v_i)$, where $C(.,.,.)$ denotes the cone defined by the three given vertices.

LEMMA 7 - CE2
Three consecutive vertices v_{i-1}, v_i, v_{i+1} of P form an ear of P iff

1. v_i is convex,

2. the closure of the triangle $\Delta(v_{i-1}, v_i, v_{i+1})$ does not contain any reflex vertex of P (except possibly v_{i-1}, v_{i+1}).

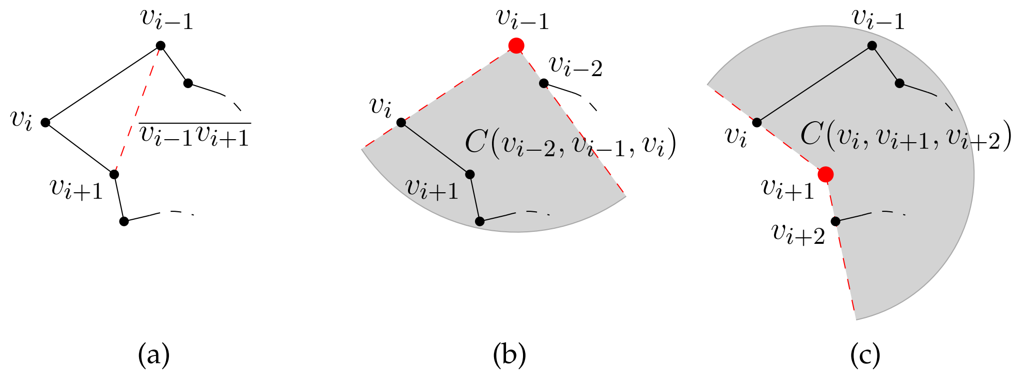


Figure 25: Visualization of the conditions needed for Lemma 6. In (a), the diagonal $\overline{v_{i-1}v_{i+1}}$, in (b), the first cone $C(v_{i-2}, v_{i-1}, v_i)$, and in (c), the second cone $C(v_i, v_{i+1}, v_{i+2})$ is shown.

Both CE1 and CE2 (Lemma 6 and Lemma 7) lead to a correct ear-clipping algorithm, as proven by Kong et al. in 1991 [KET91]. Yet the runtime analysis show that CE1 has an $\mathcal{O}(n^2)$ complexity. In the worst case, CE2 takes $\mathcal{O}(r \cdot n)$ where r denotes the number of reflex vertices [Tou91]. Both CE1 and CE2 are implemented in FIST and have been tested. In practice, since the CE2 implementation is faster than CE1, it is chosen as default.

3.1 ORIENTATION

This is a trivial problem for a simple polygon, but as we also deal with polygons which contain islands, the orientation for each contour has to be solved.

As proposed by Balbes and Siegel in [BS91], the sum of all triangle areas $\Delta(v_0, v_i, v_{i+1})$ to determine the orientation of the polygon is used [Hel01]. Since counter-clockwise (CCW) triangles have a positive value and clockwise (CW) triangles have a negative value, this gives a stable base for degenerated cases, too.

In order to decide which one is the outmost contour, one compares the absolute area-value of each contour loop and calculates it with the approach mentioned above. After the contour with the maximum absolute area is chosen for the outer loop, its orientation is set to CCW and all other contour loops are set to CW direction.

3.2 REGULAR GRID

As mentioned above, the standard ear-clipping takes $\mathcal{O}(n^2)$ time. Therefore, in practice, it is only applicable for small input data sets.

Since all reflex vertices have to be checked, the process to check whether or not a convex vertex v_j and its adjacent vertices v_{j-1} and v_{j+1} form an ear takes $\mathcal{O}(n)$ time.

A vertex v_x which would not allow v_{j-1},v_j,v_{j+1} to form an ear would have to lie inside the triangle $\Delta(v_{j-1}, v_j, v_{j+1})$ (see Figure 26a), or on the line segment $\overline{v_{j-1}v_{j+1}}$ (see Figure 26b).

This can be checked by calculating on which side v_x lies, relatively to the line segment $\overline{v_{j-1}v_{j+1}}$. Such a vertex has to be reflex, if otherwise, it can not lie inside of a triangle like $\Delta(v_{j-1}, v_j, v_{j+1})$. We can conclude that every convex polygon can be triangulated only by checking if the enclosed angle is convex. Therefore, every vertex and its adjacent vertices have to be an ear.

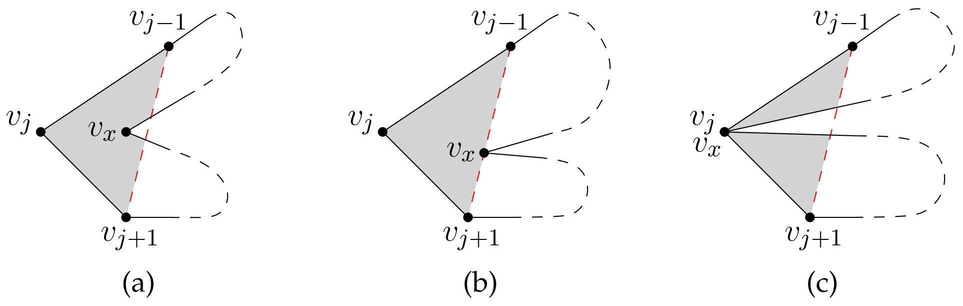


Figure 26: Intersected triangle: In (a), the triangle $\Delta(v_{j-1}, v_j, v_{j+1})$ does not form an ear as v_x lies on the inside of the diagonal $\overline{v_{j-1}v_{j+1}}$. In (b), v_x lies on the ear-defining diagonal $\overline{v_{j-1}v_{j+1}}$. Later in the triangulation, this would lead to a degenerate triangle $\Delta(v_{j-1}, v_{j+1}, v_x)$. In (c), we see a degenerate case where v_x and v_j coincide.

The idea of the regular grid is to store all reflex vertices in cells to improve this query. The dimensions are $h \cdot \sqrt{n} \times w \cdot \sqrt{n}$, where experiments showed the best results with $w \cdot h = 1$ [Hel01]. Again, in the worst case, this gives a $\mathcal{O}(n)$ query time. Since most practical input has a feasible distribution of its input vertices, the query time tends to converge to an almost constant value.

3.3 POLYGONS WITH ISLANDS

If the polygon contains islands, more than one contour loop has to be considered. A simple solution to this problem is to embed the islands into the outer contour by using bridges. A bridge consists of two diagonals which coincide but are treated separately. Due to the overlapping vertices and edges along every added bridge, the resulting polygon is not simple anymore. In Figure 27c, we see an example of such an embedding. In order to make it more visible, the co-aligned bridge edges (diagonals) have been spread apart.

There are n^2 possible bridges and each one takes $\mathcal{O}(n)$ time to verify that it is not intersecting the contour. Therefore, a naive implementation for handling islands within polygons would take $\mathcal{O}(n^3)$.

This algorithm uses the left-most vertex of an island v (see the green dots in Figure 27a) and searches for a bridge to the left v_i (see Figure 27b). As there are only n vertices as possible bridges with v, this enables a bridge finding in $\mathcal{O}(n^2)$. Again, it takes $\mathcal{O}(n)$ time to check if a vertex pair form a bridge.

PROOF - SUCH A BRIDGE ALWAYS EXISTS
Let P be a simple polygon with $n + m$ vertices and let P containing one island. Let $v_1, v_2, ..., v_n$ denote the vertices of the outer contour of P and let $i_1, i_2, ..., i_m$ denote the vertices of the island of P. Since the polygon is simple the contour of the island can not intersect the outer contour, two cases are to consider:

CASE 1 If a vertex v_i would be co-aligned with a vertex i_x we would not need a bridge, the island could already be added to the outer contour loop (also P would not be strictly simple).

CASE 2 The island-contour lies completely in $Int(P)$. If we choose the leftmost vertex of the island contour, let it be i_x, we know its interior angle must be at least π. Since Meisters showed that every simple polygon can be triangulated, we triangulate the outer contour of P without the island. Now every island vertex, and also i_x, lies on the inside of a triangle or on one of its supporting lines. This means that i_x can be connected to at least one of the vertices forming its surrounding triangle by a diagonal. This diagonal is used to create the bridge.

$\square$

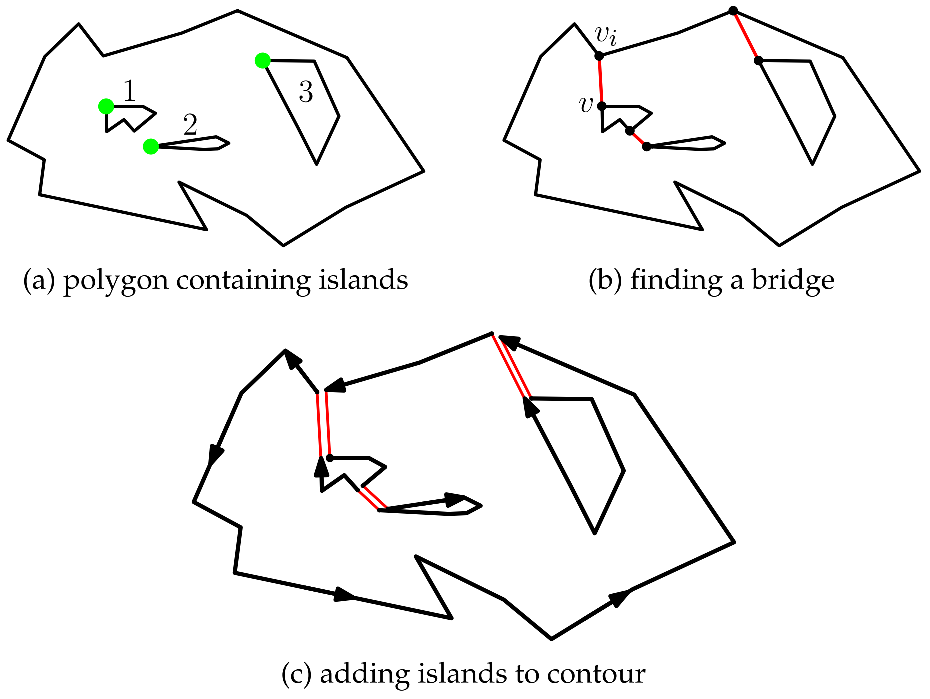
In practice, all contours are already in separate contour loops (as explained in Section 3.1). A data field is created which contains the left-most vertex from each island. These are sorted by their x-coordinate. For each of these bridge vertices, a list of possible bridges is created. Note that our data points are already sorted by their x-coordinates. This means that only the ones with a lower index as our current vertex is needed. As the vertices further to the left will not lie inside of our island-contour and since the left-most vertex is already used, only the outer contour has to be checked for intersection with a possible bridge. Those vertices are then sorted by their distance (the L_1-Norm is used). In practice, it is more likely for a closer point to be a possible bridge. That is why the sorted approach is used.

In order to speed up the search for bridges, the grid is traversed with an offset search. This means that starting with offset 0, the grid-cell in which the left-most vertex v_i of the island lies is searched. If no vertex of the outer contour lies in this grid-cell, the offset is increased to 1. Now, the search is conducted in the grid-cells above, left and below our initial cell. This process continues until a vertex v_j is found (see Figure 28) which forms a diagonal $\overline{v_j v_i}$. Since the orientation process is already completed, there has to be a vertex v_j somewhere on the left of v_i.

3.4 QUALITY TRIANGULATION

FIST is not producing an optimal triangulation like the Delaunay triangulation. To produce a quality triangulation different heuristics are implemented. A *random*, *sorted*, *top* and a *fancy* clipping variant can be used to improve the triangulation quality.

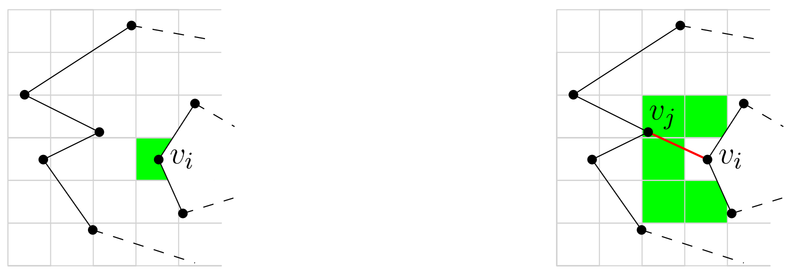
The random and sorted variant are slightly slower than the sequential variant but yield already a much "nicer" triangulation. For the sorted variant, a numerical value is calculated for each ear v_{i-1}, v_i,



(a) polygon containing islands (b) finding a bridge

(c) adding islands to contour

Figure 27: FIST: handling islands in an example polygon.



(a) Bridge finding: offset 0. (b) Bridge finding: offset 1.

Figure 28: FIST bridge finding: Further speed up due to offset search in hash grid.

v_{i+1}. This value is defined by a ratio of the diagonal $\overline{v_{i-1}v_{i+1}}$ and its enclosed angle and is scaled by the length value of the diagonal (see Figure 29b).

The fancy variant goes even further and also considers vertices close to the diagonal. In case this ear is clipped, a vertex close to the diagonal would produce sliver triangles (see Figure 29c). This approach produces an computation overhead of about 20 to 30% and offers only a slightly "better" triangulation quality than the random or sorted method.

The top variant is an improvement to the fancy approach. Using top, vertices close to the diagonal of the evaluated ear are considered as well, but no explicit search is conducted. This leads to a "better" triangulation quality than the sorted approach but without the computation overhead needed for the fancy method of 20 to 30%.

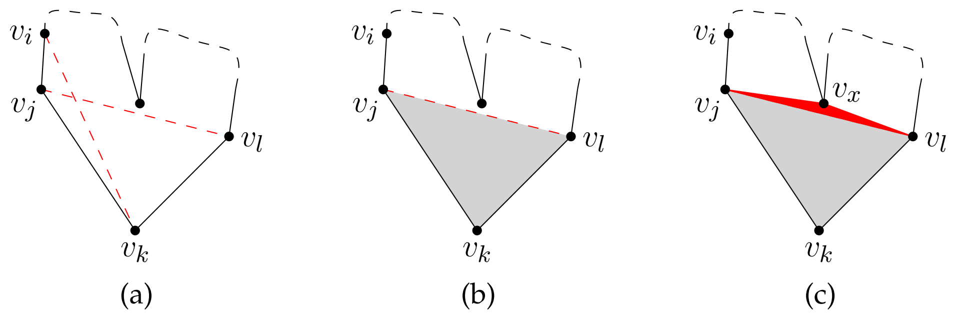


Figure 29: In (a), two possible ears are shown by their supporting diagonal. In (b), a sorted approach would favor the ear v_j, v_k, v_l before v_i, v_j, v_k. In (c), a fancy approach would also consider the vertex v_x.

4 FIST - PARALLEL

4.1 DIVIDE AND CONQUER

The first approach was to use several threads for the classification and keep the original setup of FIST. Since the classification step takes only up to 20% of the total time, we ended up with a weaker performance due to the parallelization-overhead.

We decided to search for a divide and conquer solution by splitting up the polygon into pieces. A naive implementation to find a diagonal takes $\mathcal{O}(n^3)$ time. In Section 3.3 we explained this time complexity. We tried to insert diagonals with a random approach but the finding and verification process is still too slow to be of practical relevance.

Even if we ignore the time overhead for finding diagonals we still have no guarantee that the two resulting pieces have equal amount of vertices. The vertex count of the pieces is of relevance since every thread should have the same amount of workload. We tried to avoid the implementation of a load balancer.

Due to the difficulty in finding a diagonal which lies completely in the interior of the polygon and divides the polygon into pieces of equal vertex count, we use the Sutherland-Hodgman algorithm [SH74]. It can conduct the splitting in $\mathcal{O}(n)$ time but adds possibly $\mathcal{O}(n)$ Steiner points.

We have used the Sutherland-Hodgman algorithm to split the input polygon into several pieces. The number of pieces is set to equal the number of cores which are available on the CPU in use. After the splitting, we use FIST in order to triangulate the pieces by starting each piece on a separate thread. The pieces are independent from each other which leads to a good scalability.

After the triangulation is finished we still have $\mathcal{O}(n)$ Steiner points in the data. They have to be removed, because we would add vertices which are not in the input data set and as it would also lead to a wrong triangle count.

4.1.1 *Sutherland-Hodgman Algorithm*

The algorithm provided by Sutherland and Hodgeman in 1974 can clip or split a polygon along a given split line ℓ in linear time [SH74].

The process is quit simple. We have the vertices of the polygon P in a doubly-linked list and start at a certain vertex v_i (see Figure 30a). We use two helper indices A and B to walk through the contour, starting with $A = v_i$ and $B = v_{i+1}$. We also have an empty list L where we build up the clipped polygon.

Let ℓ be our split line.

- If both A and B are above ℓ: add B to L.
- If A is above and B is below ℓ: add the intersection of $\overline{AB}$ and ℓ to L.
- And if A is below and B is above ℓ: add the intersection of $\overline{AB}$ and ℓ as well as B to L.

In Figure 30 that process is illustrated for an example polygon.

4.1.2 *Merge and Repair*

We use a separate data structure to reference the left and right contour indices of a specific Steiner point. In Figure 31 we see an example of such a split. To repair such a part we have to know its properties.

When splitting a polygon, we always use an x-coordinate which is not used inside the input data. Since all vertices are already sorted, we let $split_x = \frac{v(n/2)_x + v((n+1)/2)_x}{2}$, if $v(n/2)_x \neq v((n+1)/2)_x$. Otherwise we have to find an unequal pair. This helps us to get rid of most special cases with multiple vertices on the split line.

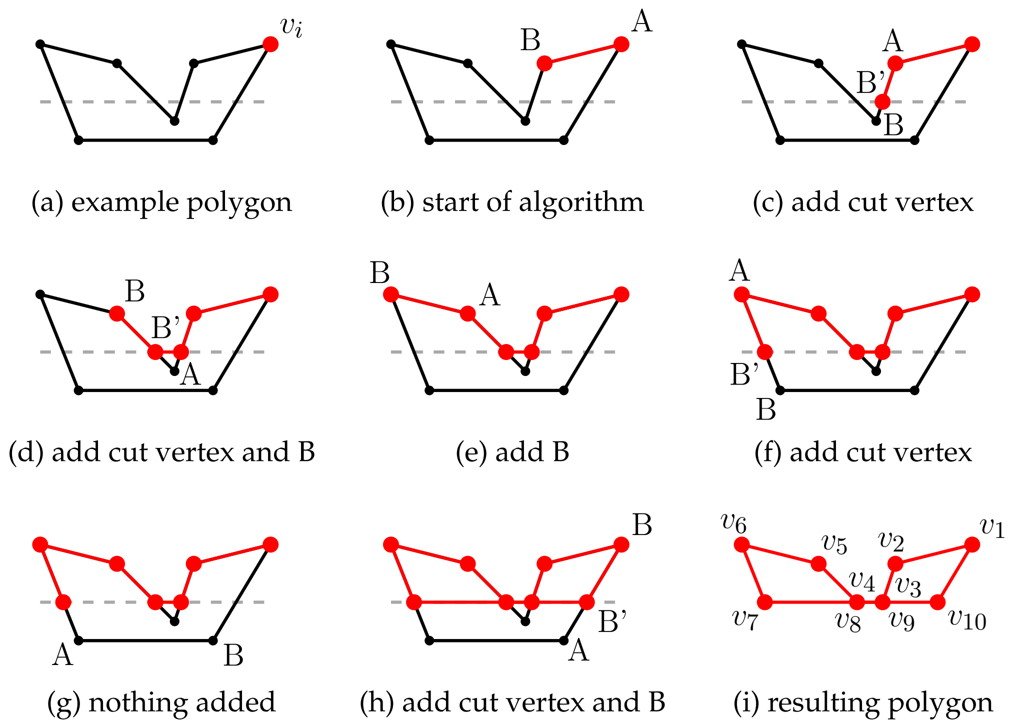


Figure 30: Sutherland-Hodgman: animation of the algorithm-process.

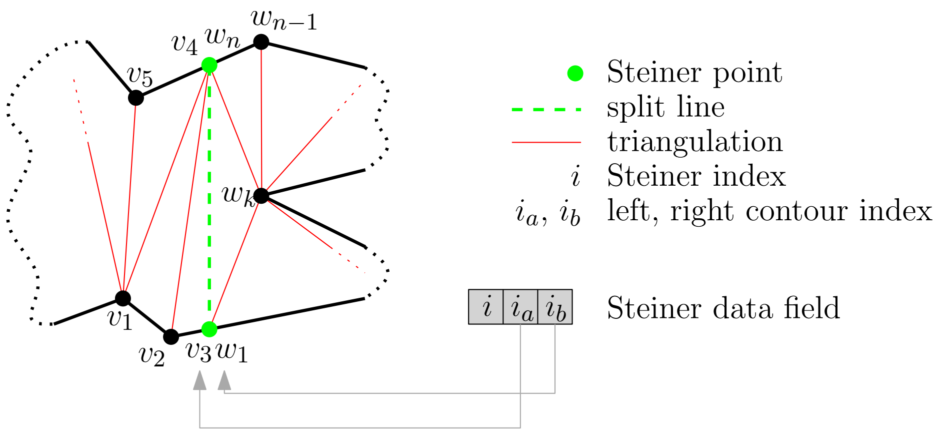


Figure 31: Example of a split and how we refer to the vertices on both sides. The Steiner data field holds the indices of the vertices of both sides for each Steiner point.

This unique $split_x$ value will result in pairwise Steiner points (s_a, s_b) which are responsible for such a split part. In order to repair such a hole we have to find and remove all related triangles on either side first. To enable this, we added triangle relationships to FIST.

As shown in Figure 32b, we remove all triangles connected to either s_a or s_b on the left and on the right side. Now we have to recreate the contour on the left side and separately on the right side (see Figure 32c). Due to the different data structures in which they are, we have no relation between the left and right side except the Steiner points.

After the contour is recreated, we fit the left and right part together and get one hole contour (see Figure 32d).

The triangulation of this hole which we create during the repair process is simpler then the standard ear-clipping which we did before. Since we remove all triangles from two Steiner points, these two vertices build the center of a "double star-shaped" polygon. This also means that each vertex, of the newly formed hole contour can be connected to at least one of the two Steiner points by a diagonal.

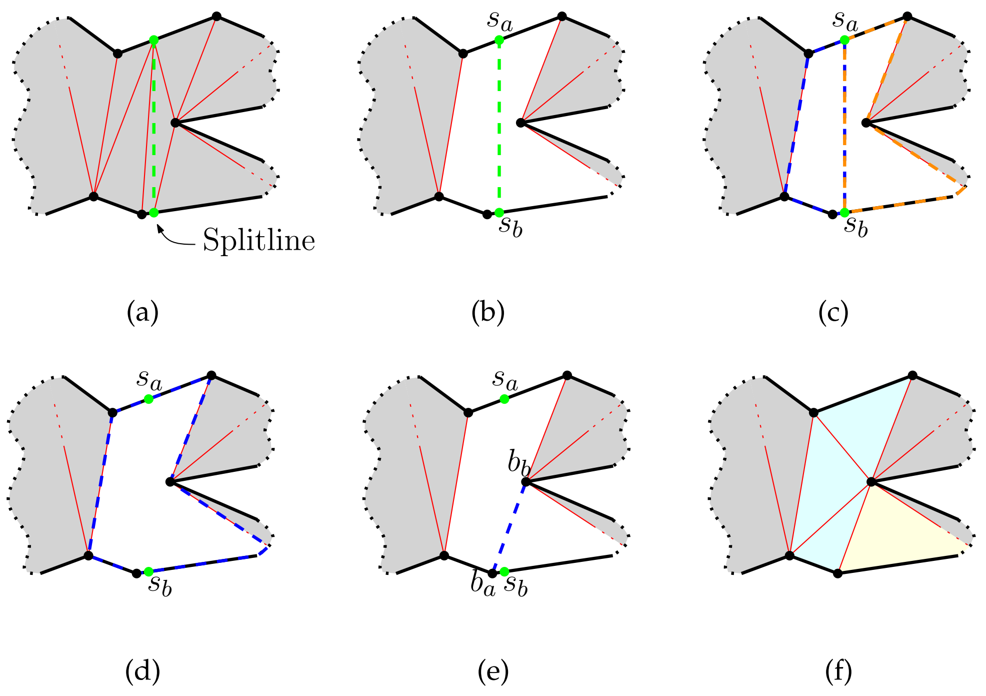


Figure 32: The repair process step by step.

DEFINITION 15 A vertex v_y is directly visible by a vertex v_x if it is possible to insert the diagonal $\overline{v_x v_y}$.

In Figure 33 we picture three scenarios. In Figure 33a we can see a contour loop of a typical split. On the right side of s_a we have several reflex vertices which are all directly visible by s_a. The reflex vertex v_r prohibits s_b to see those reflex vertices. In Figure 33b we picture an impossible situation. The red contour, yielding an unreachable area labeled B is shown. Any vertex which is contained in B would not be visible by either s_a nor s_b. This is not possible since we only removed triangles containing at least one Steiner point. Figure 33c shows a special case which we consider below.

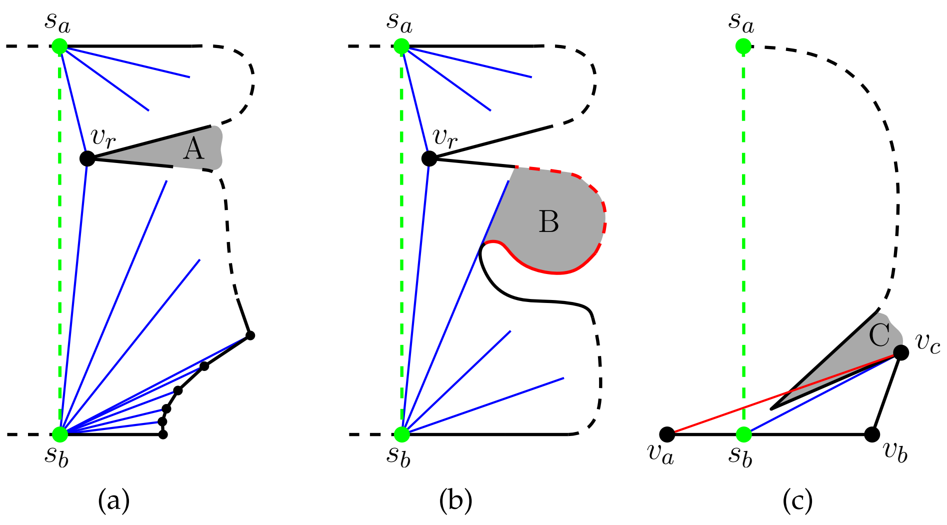


Figure 33: We see the split line as green dashed edge $\overline{s_a s_b}$. In (a), we show an example of reflex vertices contained in a contour loop. In (b), the gray area labeled A is not reachable by either Steiner point. In (c), we see an ear $s_b v_b v_c$ using a Steiner point and a line segment $\overline{v_a v_c}$ which intersects the contour. This line segment does not form a diagonal and so $v_a v_b v_c$ can not form an ear.

To get a linear complexity for the triangulation of this holes we divide the "double star-shaped" sub-polygon into two star-shaped sub-polygons. This is done by inserting a bridge into the contour loop using the following procedure:

Using one Steiner point as start vertex s_a, we walk CCW through the newly created contour. If we find a vertex which is not directly visible by s_a, we store the last visible vertex as b_a. If all vertices are visible, we stop the test when we reach the second Steiner point s_b. Then we start the same search clockwise, starting again at s_a.

The visibility check is done by an orientation test. We test a Steiner point s_a and two successive vertices v_i, v_{i+1}. If we walk CCW and the orientation is positive, then the vertex v_{i+1} is visible by s_a. If we get a negative value we store v_i as the last visible vertex in this direction.

If all vertices are visible by s_a in one direction, lets say CCW, b_a would be undefined. If we encounter an vertex which is not visible by s_a in the CW direction b_b would be defined. In such a case we define b_a as the last vertex before s_b in CCW direction.

While conducting this test in example Figure 32d we would reach s_b in the CCW search. This means that every vertex is visible by s_a. Then, in the CW search we would find b_b. As shown in Figure 32e with the blue dashed line, a bridge is inserted from b_a to b_b which splits up the contour into two contour loops, where each contains one Steiner point.

If we find neither b_a nor b_b we can triangulate directly. Otherwise we have to insert the diagonal $\overline{b_a b_b}$ and triangulate both contour parts separately (see Figure 32f).

Due to the star-shaped property of our contour we can triangulate it in linear time. We have to check that the vertex at hand is convex. Then we can classify it as an ear and clip it later.

This ear property does not hold for the two vertices next to the Steiner point. In Figure 33c we can see that v_b is a convex vertex but the three consecutive vertices v_a, v_b, v_c form no ear. This case only appears on the two vertices v_a, v_b which enclose the contour edge that also holds the Steiner point. When we triangulate, those two vertices are kept until the end. The last clipped ear v_a, v_x, v_b contains them as adjacent vertices.

Special Case

When we deal with an input contour which contains a long horizontal edge e, we will possibly split e several times. We call such an event a *multi-split* (see Figure 34a).

Since we destroy the contour list-structure when clipping ears the triangle relations are stored in the triangles themselves. After we run through the repair step, we recreate the contour for each hole and triangulate the holes again, like explained above. If two of those contour loops share an edge e, the triangle relation, between the two triangles adjacent to e, is never created. In any normal case, this relation is not needed, as the repair step is already completed.

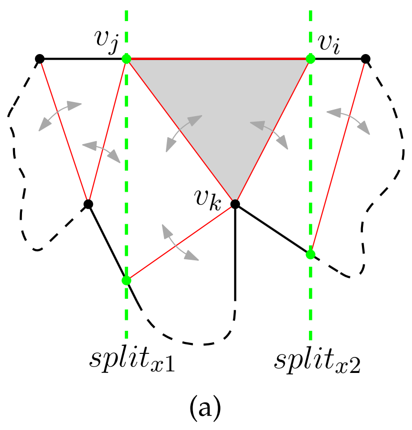
In Figure 34, we can see such a scenario where $\Delta(v_i v_j v_k)$ is the reason for the multi-split event. Figure 34b shows the already repaired second split $split_{x2}$. The lost triangle relation is marked with the red squares. The next step would be to repair $split_{x1}$, but since we can not find all triangles just by using their relations, we have to resort to a special handling.

If such a case arises, we first recreate the contour as usual. Then we start a search for triangles which belong to this split-hole as well. These triangles can be found in $\mathcal{O}(n)$ as we search only for triangles which contain one of the two split-vertices that were created by the split-line.

Due to the fact that such a case rarely occurs, the performance is hardly influenced.

4.2 MARK AND CUT

We implemented a second parallel variant of FIST. Ear-clipping by the use of a mark and cut algorithm. The idea is very simple. We start with a non-parallel mark phase followed by a parallel cut phase. For this algorithm we only need one additional data structure namely an array A to store the marked vertices.



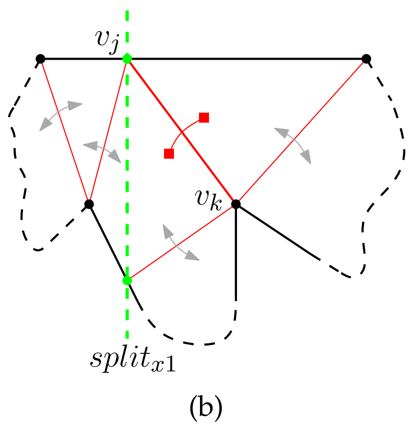


Figure 34: FIST multi-split: in (a) the triangle $\Delta(v_i v_j v_k)$ contains v_j which lies on the split-line $split_{x1}$ and v_i which lies on the split-line $split_{x2}$, in (b) $split_{x2}$ is repaired yielding a triangle relation problem marked with the red squares.

4.2.1 *Mark Phase*

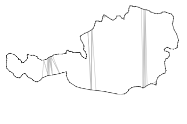
In the mark phase we walk through our contour loop once and store the index of every other vertex in A. Because we only take every other vertex, e.g. $v_2, v_4, v_6, ...$, two ears, formed by a stored vertex v_i and its two adjacent vertices v_{i-1}, v_{i+1}, can never overlap.

4.2.2 *Cut Phase*

The cut phase is conducted in parallel. For each vertex v_i in A we check whether v_{i-1}, v_i, v_{i+1} is an ear. If so, we store the triangle $\Delta(v_{i-1}, v_i, v_{i+1})$ in the triangle array at position i. The usual way FIST is storing triangles is, to add it at the last position and increment the position counter. Since a polygon of n vertices yields a triangulation with $n-2$ triangles, we would only need a array of size $n-2$ for the triangles. Now we use a preallocated array of size n and a disabled field which is set to true as a default value for each entry. Because every ear can be clipped only once and for every clipped ear only one vertex is removed from the contour (except for the last three vertices) we can use the index of that removed vertex as index for the stored triangle and avoid any collisions also in parallel.

When every vertex was checked in A we re-run the mark phase. This process is repeated until we have less than 500 vertices left in the contour loop or A contains the same amount of vertices after the mark phase. A "good" threshold has to be found through further tests. The value 500 was chosen to see the parallel behavior of this approach.

After this parallel procedure is completed we finish the last 500 vertices with the sequential variant of FIST.



5 EXPERIMENTAL RESULTS

In this chapter we will provide benchmarking results, comparing FIST using the two parallel approaches (Divide & Conquer and Mark & Cut) to the conventional variant. We explained the parallel versions of FIST in Section 4.1. To simplify the labeling we refer to the divide and conquer variant of FIST as *FIST(DC)* and to the mark and cut variant as *FIST(MC)*. All tests were performed by the use of the following test system: The cpu is an AMD Opteron$^{(TM)}$ Processor 6376 with 2.3 GHz and 64 cores. It contains 132 GB of Ram and the operating system is a Red Hat Linux with the kernel 2.6.32.

Our extensive set of over 21 000 test-samples is used to provide practical results. Samples may contain a few vertices, or up to 3 million vertices. We also differentiate between four test-sample classes: *random*, *smooth*, *thinned* and *smoother*. In Figure 35 we picture examples of these classes.

Figure 36 shows the different tasks FIST is executing. Each of the following tasks is visualized in a box-plot in percentage of total time:

INPUT Reading the input data from a file.

CLEANING Data cleaning involves a degeneration check, sorting by x-coordinate, removing of duplicates and a check whether all the contour loops are closed.

BRIDGES As explained in Section 3.3, bridges are created.

CLIPPING The actual classification and clipping step.

In the following benchmarks we only visualize the timing and speed-up for the classification and clipping step. That is because in our parallel variants of FIST we only modified this section.

Next we test FIST(DC) in the different polygon classes. In Figure 37 a, c and d we can see that FIST(DC) is faster than FIST only if the input

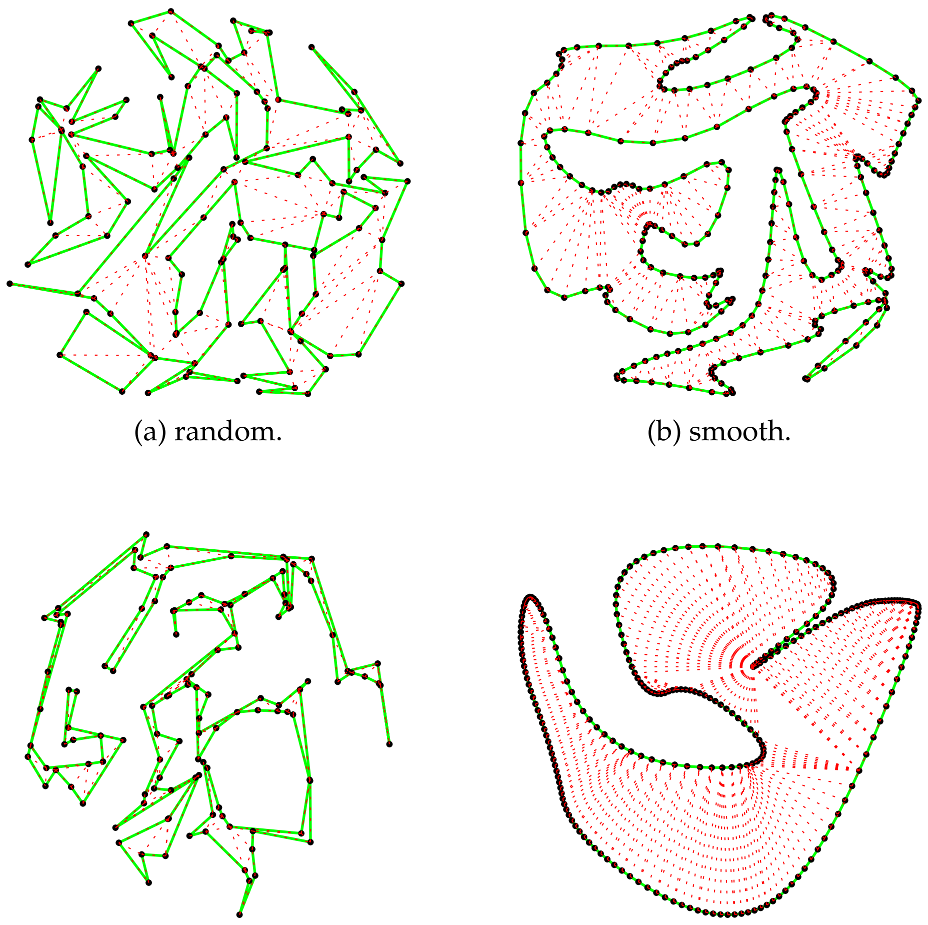

(a) random. (b) smooth. (c) thinned. (d) smoother.

Figure 35: Examples of our four polygon classes.

data exceeds about 20,000 vertices. If the input polygon contains fewer vertices, the splitting is ineffective.

The next test, Figure 38, shows the speedup of FIST(DC) compared to FIST over all our test-data. We test FIST(DC) with split sizes 2, 4, 8 and 16, which means that the test-polygon is split up into 2, 4, 8 or 16 pieces.

As one can see in Figure 38, the performance for FIST(DC 8) and FIST(DC 16) outperforms the other variants only by about 0.5. This is a poor performance as we use 8 or 16 cores to compute this outcome.

The other variants, FIST(DC 2) which has a speedup of about 1.5 and FIST(DC 4) with a speedup of 2, perform better, from cost-efficient point of few. In Figure 37b all FIST(DC) variants seem to scale better

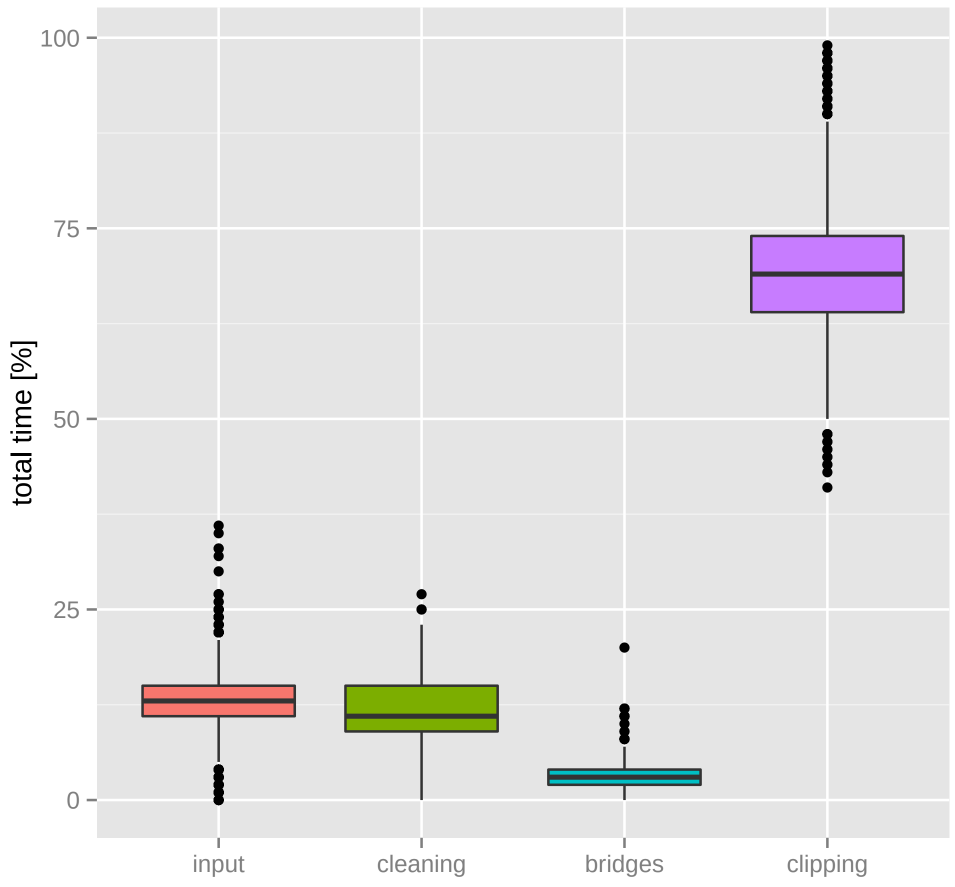
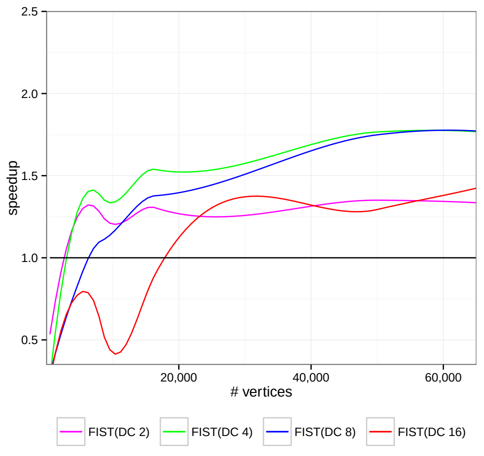


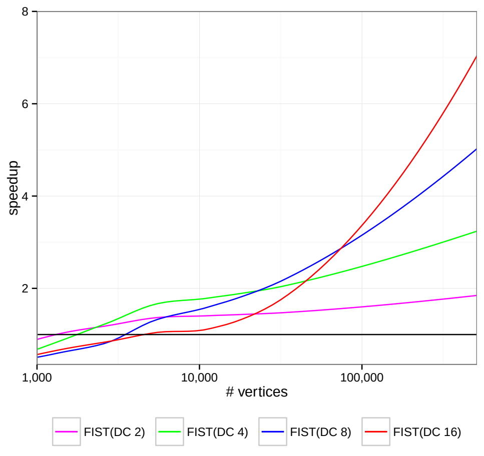
Figure 36: FIST runtime for its different tasks.

than in Figure 38. This could be explained by the character of this polygon class. The class of smooth polygons tends to be "almost" x-monotone, this leads to a fast splitting and also a fast repairing process.

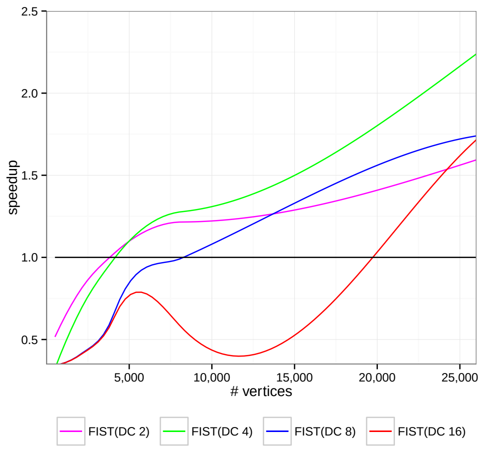
The next benchmark, Figure 39, shows the overall speedup of FIST compared to FIST(MC). We test with the use of 2, 4, 8, 16, 32 and 64 cores. The result of this test is a speedup of 1.3 up to 1.7 with a maximum of about 2. This performance is also below expectation as we use up to 64 cores and get only a speedup of 1.7.



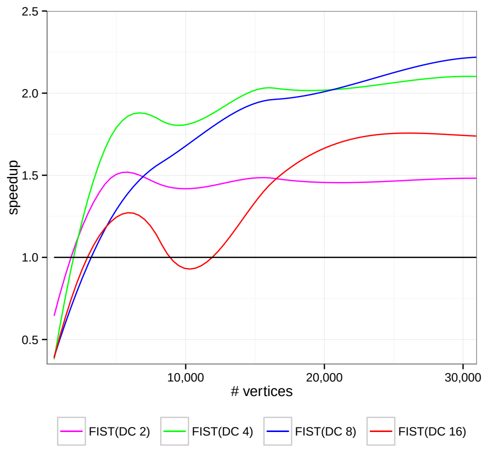
(a) Random (using 81 samples).



(b) Smooth (using 98 samples).



(c) Thinned (using 56 samples).



(d) Smoother (using 70 samples).

Figure 37: Benchmark of the four polygon classes using FIST(DC).

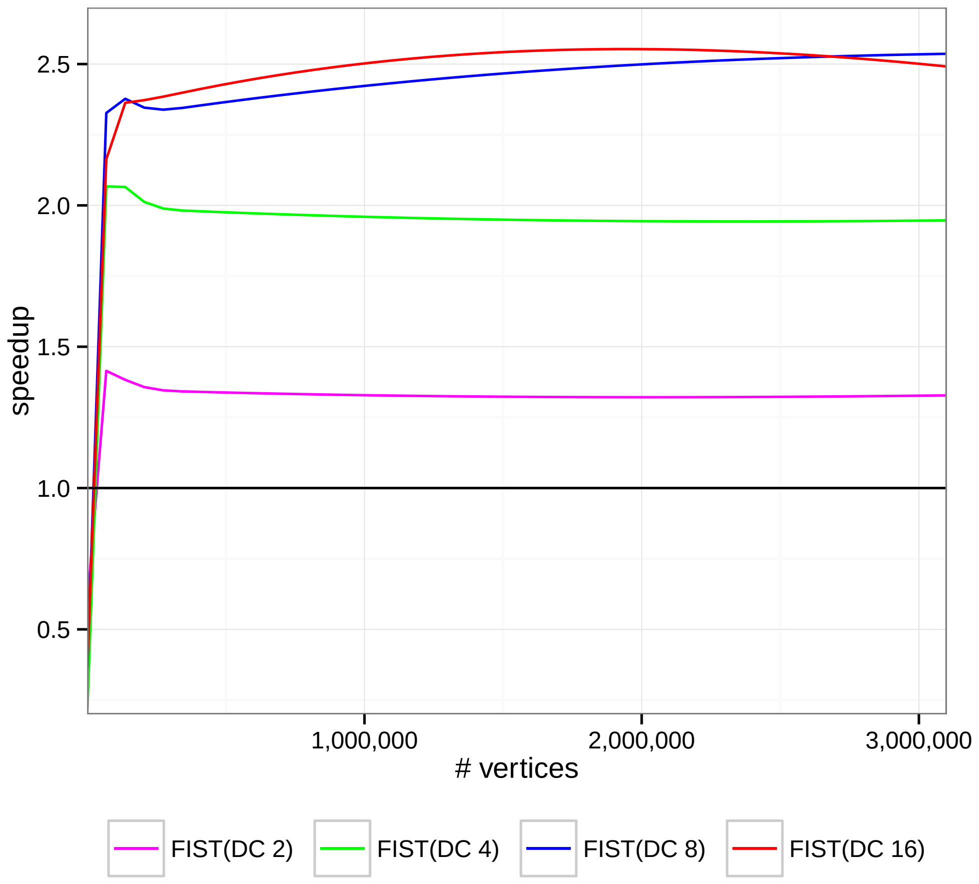


Figure 38: Benchmarking the speedup of FIST vs FIST(DC) with 2, 4, 8 and 16 sub-polygon pieces.

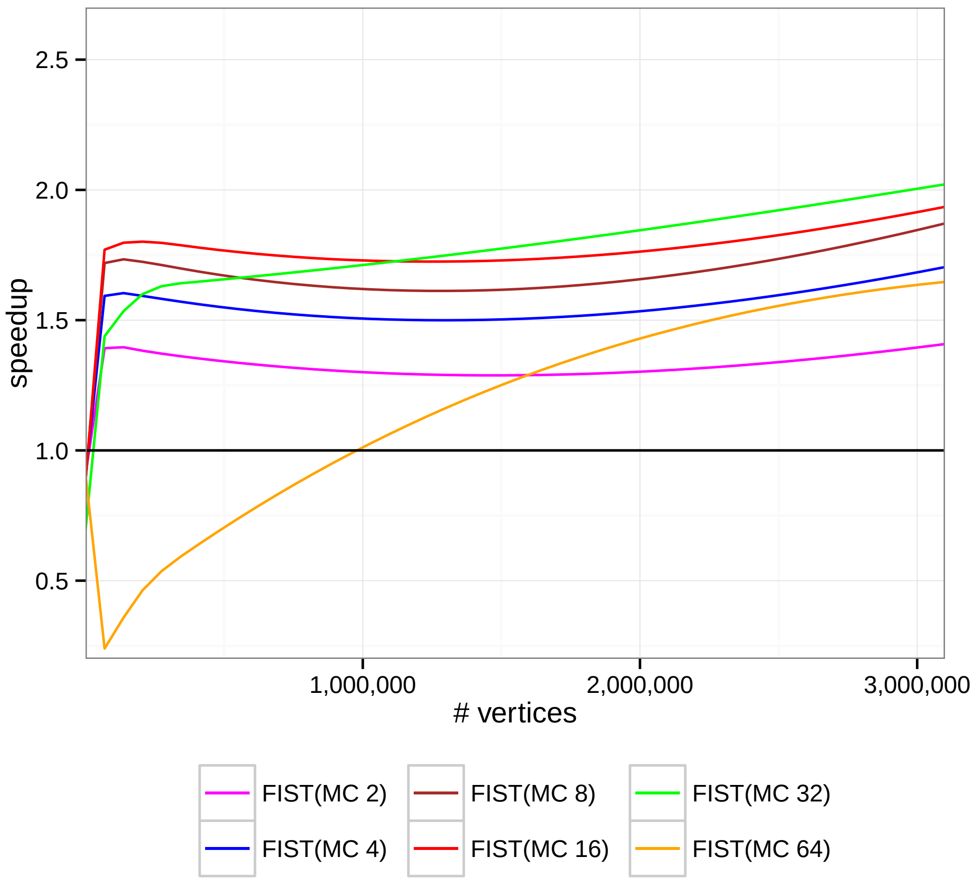


Figure 39: Benchmarking the speedup of FIST vs FIST(MC) with 2, 4, 8, 16, 32 and 64 cores.

6 CONCLUSION

The two parallel implementations of FIST, the divide and conquer approach and the mark and sweep variant, were tested in Chapter 5. Both variants of FIST turn out to result in a poor performance.

The divide and conquer approach needed a lot of fine tuning in the implementation phase and is not as fail-safe as the mark and sweep variant. Due to splitting, degenerate input or overlapping can not be handled correctly. If we compare the performance of the two implementations we can see that the speedup of FIST(DC) outperforms FIST(MC) only by about 0.5 in most cases. Only when we use a split size of 2 or at most 4 the speedup of 1.5 up to 2 is sufficient to be of practical relevance.

The idea of the mark and sweep variant is very simple and also the implementation could be accomplished with little effort. Unfortunately is a speedup of 1.7 not cost-effective as we use up to 64 cores.

A

BIBLIOGRAPHY

[Ang97] Marc Vigo Anglada. An Improved Incremental Algorithm for Constructing Restricted Delaunay Triangulations. *Computers & Graphics*, 21(2):215–223, 1997.

[BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd edition, 2008.

[Bow81] Adrian Bowyer. Computing Dirichlet Tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[BS91] R. Balbes and J. Siegel. A Robust Method for Calculating the Simplicity and Orientation of Planar Polygons. *Computer Aided Geometric Design*, 8(4):327–335, October 1991.

[CCT91] Kenneth L. Clarkson, Richard Cole, and Robert Endre Tarjan. Randomized Parallel Algorithms for Trapezoidal Diagrams. In *Seventh Annual Symposium on Computational Geometry*, pages 152–161, 1991.

[Cha91] Bernard Chazelle. Triangulating a Simple Polygon in Linear Time. *Discrete and Computational Geometry*, 6(5):485–524, 1991.

[Che89] L. P. Chew. Constrained Delaunay Triangulations. *Algorithmica*, 4(1-4):97–108, 1989.

[Che90] L. P. Chew. Building Voronoi Diagrams for Convex Polygons in Linear Expected Time. Technical report, Hanover, NH, USA, 1990.

[CTMT10] Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow Seng Tan. Parallel Banding Algorithm to Compute Exact Distance Transform with the GPU. In

Daniel G. Aliaga, Manuel M. Oliveira, Amitabh Varshney, and Chris Wyman, editors, *Symposium on Interactive 3D Graphics*, pages 83–90, 2010.

[CTW88] Kenneth L. Clarkson, Robert Endre Tarjan, and Christopher J. Van Wyk. A Fast Las Vegas Algorithm for Triangulating a Simple Polygon. In *Symposium on Computational Geometry*, pages 18–22, 1988.

[CW98] Francis Y. L. Chin and Cao An Wang. Finding the Constrained Delaunay Triangulation and Constrained Voronoi Diagram of a Simple Polygon in Linear Time. *SIAM Journal on Computing*, 28(2):471–486, 1998.

[Del34] Boris N. Delaunay. Sur la sphère vide. *Bulletin of Academy of Sciences of the USSR*, (6):793–800, 1934.

[Dv08] V. Domiter and B. Žalik. Sweep-line Algorithm for Constrained Delaunay Triangulation. *International Journal of Geographical Information Science*, 22(4):449–462, January 2008.

[For86] S. Fortune. A Sweepline Algorithm for Voronoi Diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 313–322, New York, NY, USA, 1986.

[GJPT78] M. R. Garey, David S. Johnson, Franco P. Preparata, and Robert Endre Tarjan. Triangulating a Simple Polygon. *Information Processing Letters*, 7(4):175–179, 1978.

[Gra72] Ronald L. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1(4):132–133, 1972.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[Hel01] Martin Held. FIST: Fast Industrial-Strength Triangulation of Polygons. *Algorithmica*, 30(4):563–596, 2001.

[KET91] Xianshu Kong, Hazel Everett, and Godfried Toussaint. The Graham Scan Triangulates Simple Polygons. *Pattern Recognition Letters*, 11:11–713, 1991.

[Law77] C. L. Lawson. Software for C1 Surface Interpolation. In J. R. Rice, editor, *Mathematical Software III*, pages 161–194. Academic Press, New York, 1977.

[LL86] D.T. Lee and A.K. Lin. Generalized Delaunay Triangulation for Planar Graphs. *Discrete Computational Geometry*, 1(1):201–217, 1986.

[LS80] D. T. Lee and Bruce J. Schachter. Two Algorithms for constructing a Delaunay Triangulation. *International Journal of Parallel Programming*, 9(3):219–242, 1980.

[Mei75] G. H. Meisters. Polygons have Ears. *The American Mathematical Monthly*, 82(6):648–651, June 1975.

[QCT12] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. Computing 2D Constrained Delaunay Triangulation using the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 39–46, Costa Mesa, California, 2012.

[RTCS08] Guodong Rong, Tiow-Seng Tan, Thanh-Tung Cao, and Stephanus. Computing Two-Dimensional Delaunay Triangulation using Graphics Hardware. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, pages 89–97, Redwood City, California, 2008.

[SB13] Jonathan Richard Shewchuk and Brielin C. Brown. Fast Segment Insertion and Incremental Construction of Constrained Delaunay Triangulations. In *Proceedings of the Twenty-ninth Annual Symposium on Computational Geometry*, SoCG '13, pages 299–308, Rio de Janeiro, Brazil, 2013. ACM.

[Sei91] Raimund Seidel. A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons. *Computational Geometry: Theory and Applications*, 1:51–64, 1991.

[Sei92] Raimund Seidel. Backwards Analysis of Randomized Geometric Algorithms. In *Trends in Discrete and Computational Geometry, volume 10 of Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, 1992.

[SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant Polygon Clipping. *Communications of the ACM*, 17(1):32–42, January 1974.

[SH75] Michael I. Shamos and Dan Hoey. Closest-Point Problems. In *Foundations of Computer Science, 1975, 16th Annual Symposium on*, pages 151–162. IEEE, October 1975.

[She96] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *WACG*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer, 1996.

[SSW09] Micha Sharir, Adam Sheffer, and Emo Welzl. Counting Triangulations of Planar Point Sets. *CoRR Computing Research Repository*, abs/0911.3352, 2009.

[Tou91] Godfried Toussaint. Efficient Triangulation of Simple Polygons. *The Visual Computer*, 7:280–295, 1991.

[TW88] Robert Endre Tarjan and Christopher J. Van Wyk. An $\mathcal{O}(n \log \log n)$ - Time Algorithm for Triangulating a Simple Polygon. *SIAM Journal on Computing*, 17(1):143–178, 1988.

[Vor09] G. Voronoi. Nouvelles Applications des paramètres continus à théorie des formes Quadratiques. *Journal für die reine und angewandte Mathematik*, 1909(136):67–182, January 1909.

[Ž05] Borut Žalik. An Efficient Sweep-line Delaunay Triangulation Algorithm. *Computer Aided Design*, 37(10):1027–1038, September 2005.

[Wat81] D. F. Watson. Computing the n-dimensional Delaunay Tessellation with Application to Voronoi Polytopes. *The Computer Journal*, 24(2):167–172, January 1981.

[WS85] Tony C. Woo and Sung Yong Shin. A Linear Time Algorithm for Triangulating a Point-Visible Polygon. *ACM Transactions on Graphics*, 4(1):60–69, 1985.

[XWX$^+$13] Shi-Qing Xin, Xiaoning Wang, Jiazhi Xia, Wolfgang Mueller-Wittig, Guo-Jin Wang, and Ying He. Parallel Computing 2D Voronoi Diagrams using Untransformed Sweepcircles. *Computer-Aided Design*, 45(2):483–493, 2013.

B

LIST OF FIGURES