Clipping simple polygons with degenerate intersections

Erich L. Foster · Kai Hormann · Romeo Traian Popa

Abstract

Polygon clipping is a frequent operation in many fields, including computer graphics, CAD, and GIS. Thus, efficient and general polygon clipping algorithms are of great importance. Greiner and Hormann [5] propose a simple and time-efficient algorithm that can clip arbitrary polygons, including concave and self-intersecting polygons with holes. However, the Greiner–Hormann algorithm does not properly handle degenerate intersection cases, without the undesirable need for perturbing vertices. We present an extension of the Greiner–Hormann polygon clipping algorithm that properly deals with such degenerate cases.

Citation Info

Journal
Computers & Graphics: X

Volume
2, December 2019

Pages
Article 100007, 10 pages

1 Introduction

Polygon clipping is an indispensable tool in computer graphics [4], computer aided design (CAD) [7], geographic information systems (GIS) [8], and computational sciences [3]. Applications such as VLSI circuit design [13] as well as numerical simulations typically require polygon clipping to be done thousands of times, and in GIS the polygons that are to be clipped are generally non-convex, possibly with holes and may have several thousands of vertices [12]. Therefore, efficient and general algorithms for polygon clipping are very important.

Weiler and Atherton [17] were the first to present a clipping algorithm for convex and concave polygons with holes. Their idea was developed further by Greiner and Hormann [5], who propose a simple and efficient algorithm that can also deal with self-intersecting polygons, just like Vatti's algorithm [14], which was the first to handle this most general setting.

The main advantage of the Greiner–Hormann algorithm, as compared to Vatti's algorithm, lies in its simplicity [1], but there is one serious limitation: degenerate intersections. If a vertex of one polygon lies on an edge or coincides with a vertex of the other polygon, then the algorithm fails. Greiner and Hormann suggest perturbing polygon vertices to deal with degenerate cases, which is sufficient in computer graphics, since the result remains visually correct as long as the perturbations are smaller than the size of the screen pixels. However, in most other applications the inaccuracy caused by the perturbation is undesirable.

Kim and Kim [7] present an extension of the Greiner–Hormann algorithm that deals with these degenerate cases without the need for perturbing polygon vertices. However, the method requires calculating the inside/outside status of the midpoints of all edges adjacent to an intersection, inducing a considerable additional computational cost. In the sections that follow we present an alternative approach for dealing with degeneracies that avoids these costly computations. Another, albeit less efficient method that can handle these cases is the flooding-based clipping algorithm by Wang and Manocha [15].

We start by briefly summarizing the problem (Section 2) and the original Greiner–Hormann algorithm, including its failure cases (Section 3), before presenting the proposed extensions (Section 4). In particular, the detection and classification of all possible degenerate intersections (Section 4.1) and the labelling of intersections (Section 4.2) are discussed in detail. After presenting a number of examples (Section 5), we conclude the paper with a discussion of our algorithm's advantages and limitations (Section 6).

2 Polygon clipping

Let us begin by formally defining the clipping problem. A planar *polygon* $\boldsymbol{P} = [P_1, P_2, \ldots, P_n]$ with $n \geq 3$ vertices $P_i \in \mathbb{R}^2$ is defined as the piecewise linear, closed path that is formed by joining the edges[1] $[P_1, P_2], [P_2, P_3], \ldots, [P_{n-1}, P_n], [P_n, P_1]$ that consecutively connect the vertices P_i in the given order (see Figure 1 a,c). A *complex polygon* $\mathcal{P} = \{\boldsymbol{P}_1, \boldsymbol{P}_2, \ldots, \boldsymbol{P}_m\}$ is a set of $m \geq 1$ polygons $\boldsymbol{P}_j$, called the *components* of $\mathcal{P}$ (see Figure 1 b,d). We follow the convention that the interior of $\mathcal{P}$ is determined by the *even-odd rule* [4] and consists of all points $p \in \mathbb{R}^2$ which do not lie on any of the edges of $\mathcal{P}$ and for which a ray drawn

[1] Without loss of generality, we assume successive vertices to be distinct, so that the half-open edges $[P_i, P_{i+1})$ are not empty.

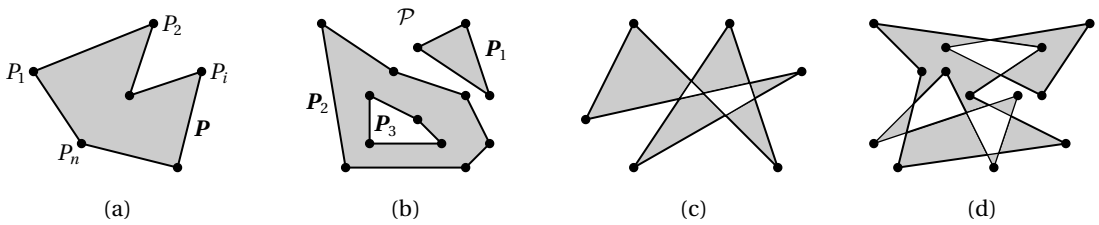


Figure 1: Examples of a single simple polygon (a), a simple complex polygon with three components and one hole (b), a single self-intersecting polygon (c), and a complex self-intersecting polygon with three components (d), with their interiors shaded.

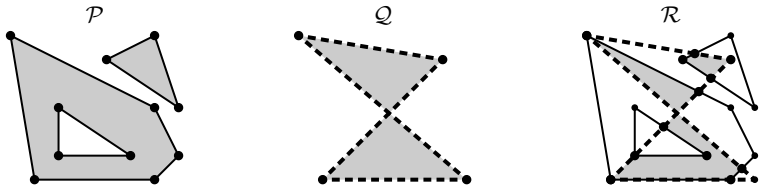


Figure 2: Example of polygon clipping: the intersection of a simple polygon $\mathcal{P}$ with three components and one hole (a) and a self-intersecting polygon $\mathcal{Q}$ with one component (b) gives a result polygon $\mathcal{R}$ with two components, one of them simple, the other self-intersecting (c).

from p to infinity in any direction crosses $\mathcal{P}$ an odd number of times. Because of this definition, components that are inside other components are commonly referred to as *holes* (see Figure 1 b). For the sake of brevity, we consider single polygons as complex polygons with one component and refer to complex polygons simply as polygons. A polygon is called *simple* if it does not cross itself, that is, its edges intersect only at common endpoints, which in turn is equivalent to the property that its half-open edges do not intersect at all. Each component of a simple polygon is thus topologically equivalent to a circle (see Figure 1 a,b).

Polygon clipping usually refers to computing the *intersection* $\mathcal{P} \cap \mathcal{Q}$ of the interiors of two polygons $\mathcal{P}$ and $\mathcal{Q}$, often called the *clip* and the *subject* polygon, which is itself a region bounded by a polygon $\mathcal{R}$ (see Figure 2). Most clipping algorithms can be modified to also compute other polygon set operations, like the *union* $\mathcal{P} \cup \mathcal{Q}$ and the differences $\mathcal{P} \setminus \mathcal{Q}$ and $\mathcal{Q} \setminus \mathcal{P}$. Especially in computer graphics, polygon clipping may also refer more specifically to the process of fragmenting the subject polygon into those parts that lie inside the clip polygon and those that lie outside the clip polygon [16]. However, we follow the more common convention that clipping $\mathcal{P}$ and $\mathcal{Q}$ yields $\mathcal{P} \cap \mathcal{Q}$ and note that the result is symmetric with respect to $\mathcal{P}$ and $\mathcal{Q}$.

3 Greiner–Hormann algorithm

The Greiner–Hormann polygon clipping algorithm [5] consists of three phases. The *intersection phase* computes all intersections points between $\mathcal{P}$ and $\mathcal{Q}$ and inserts them as new vertices into both polygons. In Figure 3, there are eight such intersection points $I_1, \ldots, I_8$, and the algorithm adds, for example, I_1 as a new vertex of $\mathcal{P}$ between P_1 and P_2 and I_1, I_6, I_8, and I_3 in this order as new vertices of $\mathcal{Q}$ between Q_3 and Q_4. Greiner and Hormann propose to represent all polygon components with circular doubly-linked lists to facilitate the vertex insertion operation and to link corresponding pairs of intersection vertices using additional neighbour pointers that are needed in the third phase (see Figure 4).

The *labelling phase* marks each intersection vertex I of $\mathcal{P}$ as entry or exit point, depending on whether someone travelling along $\mathcal{P}$ in the given order enters or leaves the interior of $\mathcal{Q}$ at I, and similarly for the intersection vertices of $\mathcal{Q}$. To this end, the algorithm starts for each component $\boldsymbol{P}$ of $\mathcal{P}$ at the first vertex P of $\boldsymbol{P}$, determines whether P lies inside or outside $\mathcal{Q}$ [6], and then traverses all vertices of $\boldsymbol{P}$ in the given order, labelling the intersection vertices alternately as entry or exit. For the example in Figure 3, the first vertex P_1 of the first component of $\mathcal{P}$ is identified as lying inside $\mathcal{Q}$, so that the next intersection vertex along this component, I_1, is marked as exit, the second next, I_2, as entry, and so on. For the second component of $\mathcal{P}$, its first vertex P_6 is found to lie outside $\mathcal{Q}$, hence I_5 gets an entry and I_6 an exit label, etc. For $\mathcal{Q}$, the algorithm determines that Q_1 lies outside $\mathcal{P}$ and then marks I_2, this time as a vertex of $\mathcal{Q}$, as entry, then I_7 as exit, and so forth.

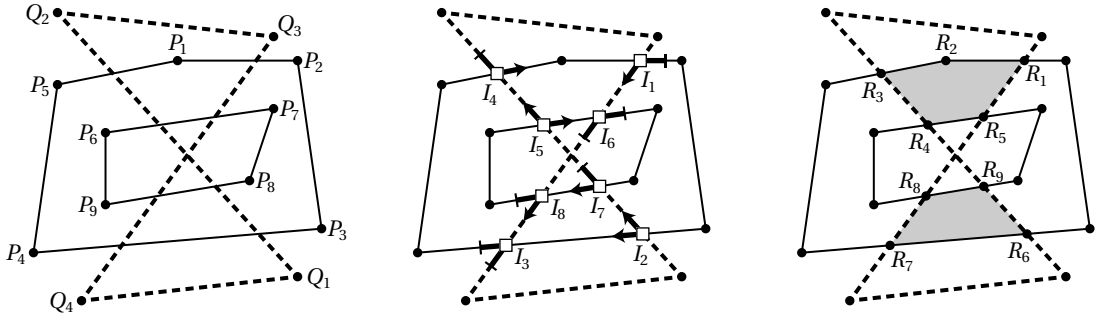


Figure 3: For two given polygons $\mathcal{P}$ and $\mathcal{Q}$ (a), the Greiner–Hormann algorithm first computes all intersection points (□), then marks them as entry (➞) or exit (⊣) points for both polygons (b), and finally generates the result polygon $\mathcal{R}$ (c).

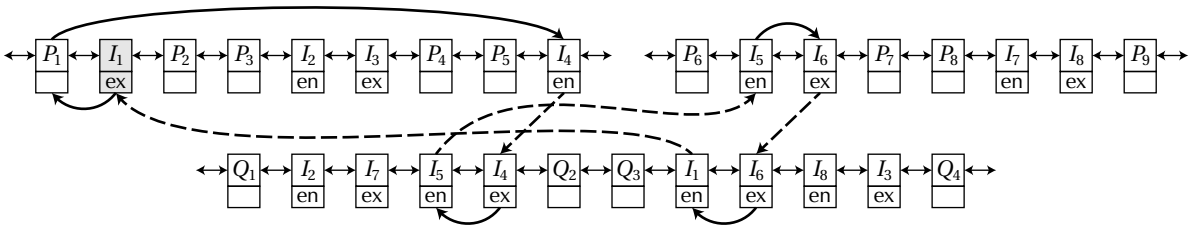


Figure 4: The Greiner–Hormann algorithm uses doubly-linked lists to represent polygon components. After inserting and labelling the intersection vertices (cf. Figure 3 b), each component $\boldsymbol{R}$ of $\mathcal{R}$ is traced out by starting at an intersection vertex on $\mathcal{P}$ (shaded), moving along $\mathcal{P}$ in the correct order, switching over to $\mathcal{Q}$ at the next intersection vertex, and repeating this process until $\boldsymbol{R}$ is closed. The switching step requires linking corresponding intersection vertices of $\mathcal{P}$ and $\mathcal{Q}$ with neighbour pointers (dashed).

The *tracing phase* finally generates all components of the result polygon $\mathcal{R}$. Starting at an intersection vertex I of $\mathcal{P}$, the algorithm moves along the corresponding component of $\mathcal{P}$ either in the forward or backward direction, depending on whether the label of I is entry or exit, respectively, until the next intersection vertex is encountered. Using the neighbour pointer, the algorithm then switches to the corresponding intersection vertex of $\mathcal{Q}$ and repeats this process until it returns to I. All vertices visited this way and in this order constitute one component of $\mathcal{R}$, and the algorithm continues generating components until all intersection vertices have been visited. For the example in Figure 3, the tracing of the first component starts at I_1. Since I_1 is marked as an exit point, we traverse $\mathcal{P}$ backward, encountering first P_1 and then I_4, where we switch over to $\mathcal{Q}$. As a vertex of $\mathcal{Q}$, the label of I_4 is exit, hence we proceed backwards to I_5 and switch back to $\mathcal{P}$. Observing that I_5 on $\mathcal{P}$ is an entry point, we advance forward to I_6, and after switching, moving along $\mathcal{Q}$ to I_1, and switching back, we arrive at the initial vertex I_1 on $\mathcal{P}$. This completes the tracing of the first component $\boldsymbol{R}_1$ of $\mathcal{R}$ with vertices $R_1 = I_1$, $R_2 = P_1$, $R_3 = I_4$, $R_4 = I_5$, $R_5 = I_6$ (see Figures 3 c and 4). After generating the second component $\boldsymbol{R}_2 = [R_6, R_7, R_8, R_9] = [I_2, I_3, I_8, I_7]$ in the same way with I_2 on $\mathcal{P}$ as the initial vertex, all intersection vertices have been visited and the algorithm terminates.

3.1 Degeneracies

Despite its favourable simplicity, a serious limitation of the Greiner–Hormann algorithm is that it cannot deal with degenerate intersections, that is, if a vertex of $\mathcal{P}$ lies on an edge or coincides with a vertex of $\mathcal{Q}$ or vice versa. For example, if P_3 in Figure 5 a is detected as an intersection and hence inserted as an intersection vertex into both $\mathcal{P}$ and $\mathcal{Q}$, then the result will be incorrect, because the strategy of labelling intersection vertices alternately as entry and exit gives wrong labels in this case. While the problem may be fixed by slightly perturbing any such degenerate intersection vertices, the method of perturbation can result in different solutions depending upon the perturbation direction. In the previous example, moving P_3 slightly towards the interior of $\mathcal{Q}$ gives a result polygon with five vertices (see Figure 5 b), and any perturbation in the opposite direction produces an intersection polygon with two triangular components (see Figure 5 c). This renders the perturbation method indeterminate and not appropriate for various applications, such as numerical simulation [3].

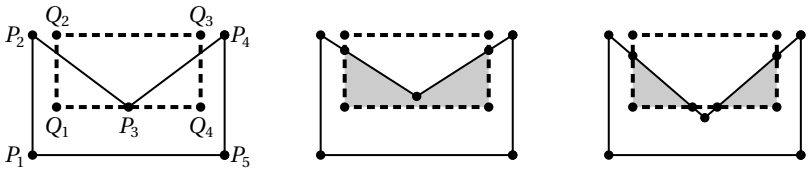


Figure 5: The Greiner–Hormann algorithm cannot deal with degenerate intersections like the one at P_3 (a), and while perturbing the vertex helps to overcome this limitation, different perturbation directions may lead to geometrically and topologically different results (b, c).

4 Extension of the Greiner–Hormann algorithm

Inspired by the work of Kim and Kim [7], we figured out that the aforementioned limitation of the Greiner–Hormann algorithm can be overcome with rather small changes that do not affect the simplicity of the algorithm. In fact, our extension mainly requires a more refined analysis of the intersection vertices in the labelling phase of the algorithm, provided that degenerate intersections are detected and handled correctly in the intersection phase. Our revised labelling strategy uses local orientation tests to identify and mark a subset of all intersection vertices as crossing intersections. These are then labelled alternately as entry and exit points exactly as in the original Greiner–Hormann algorithm, and also the tracing phase of the algorithm remains basically the same.

4.1 Intersection phase

The first phase of our algorithm is essentially the same as in the Greiner–Hormann algorithm as it finds all intersection points of $\mathcal{P}$ and $\mathcal{Q}$, but we must deal with degenerate intersections appropriately. To this end, we test the half-open edges of $\mathcal{P}$ against the half-open edges of $\mathcal{Q}$ for potential intersections, so as to avoid detecting a possible intersection at a vertex twice. Without loss of generalization, let us consider the half-open edges $[P_1, P_2)$ and $[Q_1, Q_2)$ and distinguish two cases.

If both edges are not parallel, then there exists a unique intersection point of the two lines defined by both edges,

$$I = (1-\alpha)P_1 + \alpha P_2 = (1-\beta)Q_1 + \beta Q_2, \qquad \alpha, \beta \in \mathbb{R},$$

and the edges themselves intersect at I, if and only if $0 \leq \alpha, \beta < 1$. The parameters α and β describe the relative position of I between P_1 and P_2 and between Q_1 and Q_2, respectively. They can be determined as

$$\begin{aligned} \alpha &= \frac{\mathrm{A}(P_1, Q_1, Q_2)}{\mathrm{A}(P_1, Q_1, Q_2) - \mathrm{A}(P_2, Q_1, Q_2)}, \\ \beta &= \frac{\mathrm{A}(Q_1, P_1, P_2)}{\mathrm{A}(Q_1, P_1, P_2) - \mathrm{A}(Q_2, P_1, P_2)}, \end{aligned} \tag{1}$$

where

$$\mathrm{A}(P, Q, R) = (Q_x - P_x)(R_y - P_y) - (Q_y - P_y)(R_x - P_x),$$

is the function that computes twice the signed area of the triangle $[P, Q, R]$. Note that the denominators in (1) do not vanish as long as $[P_1, P_2)$ and $[Q_1, Q_2)$ are not parallel. We classify the possible intersection types as shown in Figure 6:

- **X-intersection**: this non-degenerate intersection occurs if and only if $0 < \alpha, \beta < 1$. In this case, we add I to $\mathcal{P}$ and $\mathcal{Q}$ and link the two copies with the neighbour pointer as described in [5].
- **T-intersection**: if $\alpha = 0$ and $0 < \beta < 1$, then P_1 lies on the edge $[Q_1, Q_2]$, but does not coincide with Q_1 or Q_2. In this case, we add a copy of P_1 to $\mathcal{Q}$ and link it with P_1. Likewise, a copy of Q_1 is added to $\mathcal{P}$ and linked with Q_1, if $\beta = 0$ and $0 < \alpha < 1$.
- **V-intersection**: if $\alpha = \beta = 0$, then both edges intersect at $P_1 = Q_1$, and we link P_1 with Q_1.

We do not consider degenerate intersection cases involving P_2, because they will be detected as soon as we move on to the next edge $[P_2, P_3)$ of $\mathcal{P}$, and the same holds for degenerate cases involving Q_2.

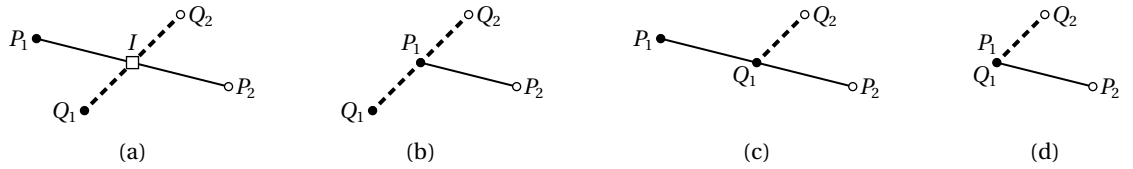


Figure 6: Possible intersection types for two non-parallel edges $[P_1, P_2)$ and $[Q_1, Q_2)$: *X-intersection* (a), *T-intersection* (b, c), and *V-intersection* (d). Note that P_2 and Q_2 are depicted by empty circles to emphasize that we are considering half-open edges.

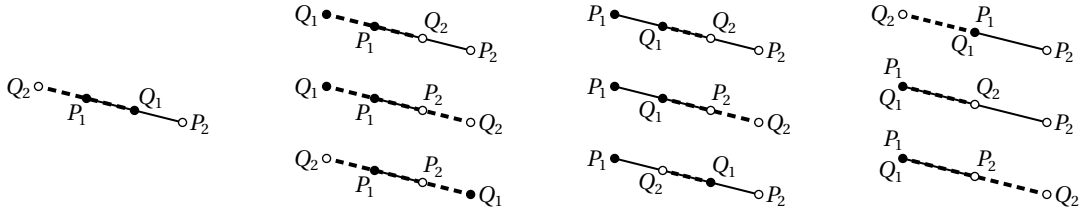


Figure 7: Possible overlap types for two collinear edges $[P_1, P_2)$ and $[Q_1, Q_2)$: *X-overlap* (a), *T-overlap* (b, c), and *V-overlap* (d).

If both edges are parallel, then they can intersect or rather overlap only if they are collinear, that is, if

$$\mathrm{A}(P_1, Q_1, Q_2) = \mathrm{A}(P_2, Q_1, Q_2) = \mathrm{A}(Q_1, P_1, P_2) = \mathrm{A}(Q_2, P_1, P_2) = 0.$$

Under this assumption, we can express Q_1 relative to $[P_1, P_2)$ and P_1 relative to $[Q_1, Q_2)$ as

$$Q_1 = (1-\alpha)P_1 + \alpha P_2, \qquad P_1 = (1-\beta)Q_1 + \beta Q_2,$$

and the parameters α and β can be determined as

$$\alpha = \frac{\langle Q_1 - P_1, P_2 - P_1 \rangle}{\langle P_2 - P_1, P_2 - P_1 \rangle}, \qquad \beta = \frac{\langle P_1 - Q_1, Q_2 - Q_1 \rangle}{\langle Q_2 - Q_1, Q_2 - Q_1 \rangle},$$

where $\langle \cdot, \cdot \rangle$ denotes the standard dot product in $\mathbb{R}^2$. We classify the possible overlap types in analogy to the intersection types above and as shown in Figure 7:

- **X-overlap**: this type of overlap occurs if and only if $0 < \alpha, \beta < 1$. In this case, we add a copy of P_1 to $\mathcal{Q}$, linked with P_1, and a copy of Q_1 to $\mathcal{P}$, linked with Q_1.
- **T-overlap**: if $\alpha < 0$ or $\alpha \geq 1$, and $0 < \beta < 1$, then we add a copy of P_1 to $\mathcal{Q}$, linked with P_1. Likewise, a copy of Q_1, linked with Q_1, is added to $\mathcal{P}$, if $\beta < 0$ or $\beta \geq 1$, and $0 < \alpha < 1$.
- **V-overlap**: if $\alpha = \beta = 0$, then $P_1 = Q_1$, and we link P_1 with Q_1.

Again, we do not consider overlap cases involving P_2 or Q_2, for the same reasons as above.

After executing the first phase of our algorithm, it is guaranteed that all intersections of $\mathcal{P}$ and $\mathcal{Q}$ occur at common intersection vertices, which are linked by neighbour pointers, or along common segments, which are now represented as common edges in both $\mathcal{P}$ and $\mathcal{Q}$ with common intersection vertices as endpoints. Figure 8 shows an example.

4.2 Labelling phase

As in the Greiner–Hormann algorithm, the goal of the second phase is to mark the previously found intersection vertices as entry or exit points. If all intersections of $\mathcal{P}$ and $\mathcal{Q}$ are assumed to be non-degenerate *X-intersections*, then marking these vertices is simple, because an entry intersection vertex is always followed by an exit intersection vertex and vice versa (see Section 3). Degenerate intersections, however, require a more careful investigation of the local situation around each intersection vertex.

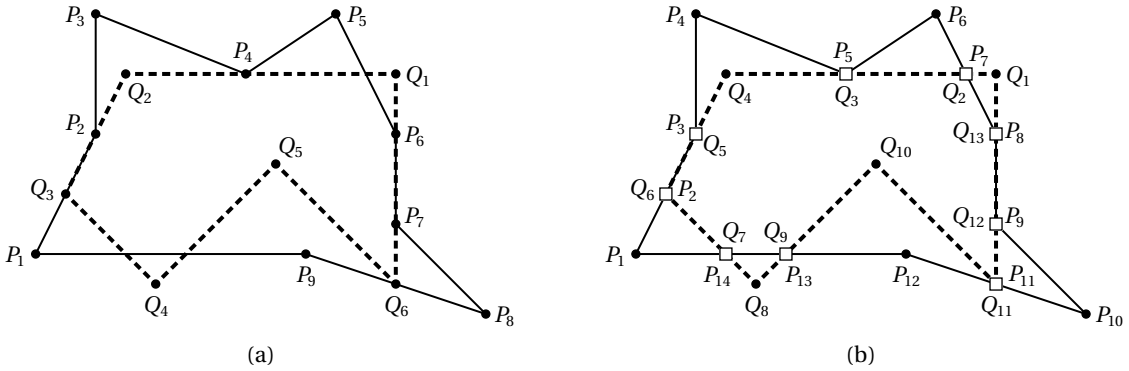


Figure 8: Example of two polygons before (a) and after (b) executing the intersection phase. Note that we renumber the vertices of both polygons in (b) to simplify the notation. The algorithm detects nine common intersection vertices (□) and the two common segments of $\mathcal{P}$ and $\mathcal{Q}$ are now represented as edges $[P_2,P_3]=[Q_5,Q_6]$ and $[P_8,P_9]=[Q_{12},Q_{13}]$.

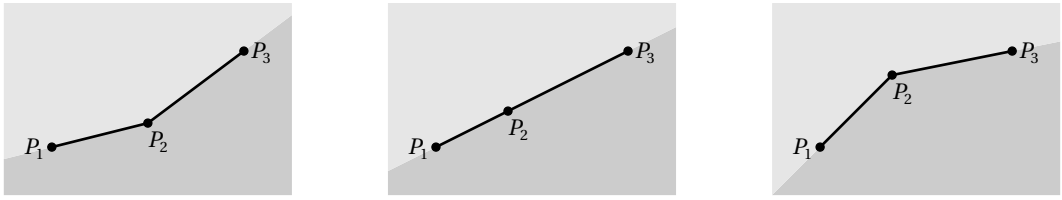


Figure 9: Regions to the left (light grey) and to the right (dark grey) of the polygonal chain (P_1,P_2,P_3) for the three possible cases: *left turn* (a), *straight* (b), and *right turn* (c).

To this end, let us first recall that a point Q lies to the left of the edge $[P_1,P_2]$ if $\mathrm{A}(Q,P_1,P_2)>0$ and to the right if $\mathrm{A}(Q,P_1,P_2)<0$. If we now consider two adjacent edges $[P_1,P_2]$ and $[P_2,P_3]$, then we can determine whether Q lies to the left or to the right of the polygonal chain (P_1,P_2,P_3) by computing

$$s_1=\mathrm{A}(Q,P_1,P_2),\quad s_2=\mathrm{A}(Q,P_2,P_3),\quad s_3=\mathrm{A}(P_1,P_2,P_3)$$

and distinguishing three cases as shown in Figure 9:

- **Left turn**: if $s_3>0$, then the chain takes a left turn at P_2 and Q lies to the left of (P_1,P_2,P_3) if $s_1>0$ and $s_2>0$, and to the right if $s_1<0$ or $s_2<0$.
- **Straight**: if $s_3=0$, then $\mathrm{sign}(s_1)=\mathrm{sign}(s_2)$ and Q lies to the left of (P_1,P_2,P_3) if $s_1>0$ and to the right if $s_1<0$.
- **Right turn**: if $s_3<0$, then the chain takes a right turn at P_2 and Q lies to the left of (P_1,P_2,P_3) if $s_1>0$ or $s_2>0$, and to the right if $s_1<0$ and $s_2<0$.

Clearly, the case of a straight polygonal chain can be included in either of the other two cases, for the sake of simplifying the code.

Now let I be an intersection vertex of $\mathcal{P}$, preceded by P_- and succeeded by P_+. As a consequence of the first phase, I is also a vertex of $\mathcal{Q}$ with neighbours Q_- and Q_+. We then distinguish two possible cases.

If the four edges adjacent to I do not overlap, then the local behaviour of $\mathcal{P}$ with respect to $\mathcal{Q}$ at I can be classified as shown in Figure 10:

- **Crossing**: if Q_- and Q_+ lie on different sides of (P_-,I,P_+), then $\mathcal{P}$ crosses $\mathcal{Q}$ at I, and we mark I as crossing.
- **Bouncing**: if Q_- and Q_+ lie on the same side of (P_-,I,P_+), then $\mathcal{P}$ does not cross $\mathcal{Q}$ at I, and we mark I as bouncing.

For the example in Figure 8 b, this classification scheme marks P_7, P_{13}, P_{14} as crossing and P_5, P_{11} as bouncing (see Figure 12 a).

The situation is slightly more complicated, if I is the endpoint of a common segment. If the edge $[I,P_+]$ of $\mathcal{P}$ overlaps with $\mathcal{Q}$, then it is either equal to $[Q_-,I]$ or $[I,Q_+]$, because all common segments are represented

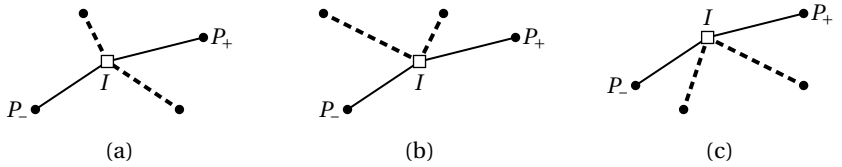


Figure 10: Possible local configurations without overlaps around an intersection vertex I after the first phase: *crossing* (a) and *bouncing* (b, c).

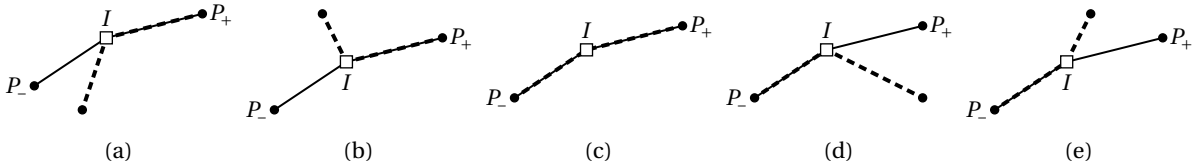


Figure 11: Possible local configurations with overlaps around an intersection vertex I of $\mathcal{P}$ with respect to $\mathcal{Q}$ after phase one: *left/on* (a) *right/on* (b), *on/on* (c), *on/left* (d), *on/right* (e).

as common edges after phase one. Therefore, this situation can be detected by checking if P_+ is itself an intersection vertex and linked to either Q_- or Q_+, and a similar test reveals if the edge $[P_-, I]$ of $\mathcal{P}$ overlaps with $\mathcal{Q}$. With these considerations in mind, we can distinguish the five cases shown in Figure 11 for describing the local position of $\mathcal{P}$ around I relative to $\mathcal{Q}$:

- **Left/On**: if P_+ is linked to Q_+ (or Q_-) and Q_- (or Q_+) lies to the right of (P_-, I, P_+), then $\mathcal{P}$ changes from being left of $\mathcal{Q}$ to being on $\mathcal{Q}$ at I.
- **Right/On**: if P_+ is linked to Q_+ (or Q_-) and Q_- (or Q_+) lies to the left of (P_-, I, P_+), then $\mathcal{P}$ changes from being right of $\mathcal{Q}$ to being on $\mathcal{Q}$ at I.
- **On/On**: if P_+ is linked to Q_+ (or Q_-) and P_- is linked to Q_- (or Q_+), then $\mathcal{P}$ is on $\mathcal{Q}$ to both sides of I.
- **On/Left**: if P_- is linked to Q_- (or Q_+) and Q_+ (or Q_-) lies to the right of (P_-, I, P_+), then $\mathcal{P}$ changes from being on $\mathcal{Q}$ to being left of $\mathcal{Q}$ at I.
- **On/Right**: if P_- is linked to Q_- (or Q_+) and Q_+ (or Q_-) lies to the left of (P_-, I, P_+), then $\mathcal{P}$ changes from being on $\mathcal{Q}$ to being right of $\mathcal{Q}$ at I.

After this analysis, all intersection vertices of $\mathcal{P}$ with adjacent overlapping edges form polygonal intersection chains $\boldsymbol{I} = (I_1, I_2, \ldots, I_k)$ with $k > 1$, where I_1 is marked as *x/on*, $I_2, \ldots, I_{k-1}$ are marked as *on/on*, and I_k is marked as *on/y* with $x, y \in \{\textit{left}, \textit{right}\}$. Each polygonal intersection chain $\boldsymbol{I}$ can then be classified as follows:

- **Delayed crossing**: if $x \neq y$, then $\mathcal{P}$ crosses $\mathcal{Q}$ at $\boldsymbol{I}$. In this case, we mark the intersection vertices $I_1, \ldots, I_{k-1}$ as bouncing and I_k as crossing.
- **Delayed bouncing**: if $x = y$, then $\mathcal{P}$ does not cross $\mathcal{Q}$ at $\boldsymbol{I}$, and we mark all intersection vertices $I_1, \ldots, I_k$ as bouncing.

Note that in the case of a *delayed crossing* we could actually mark any intersection vertex in $\boldsymbol{I}$ as crossing, as long as all other vertices in $\boldsymbol{I}$ are marked as bouncing. For the example in Figure 8, the strategy above identifies a *delayed crossing* at (P_8, P_9) and a *delayed bouncing* at (P_2, P_3), and consequently marks P_9 as crossing and P_2, P_3, P_8 as bouncing (see Figure 12 a).

Once the intersection vertices of $\mathcal{P}$ have been marked as crossing or bouncing, we can simply copy these labels to the intersection vertices of $\mathcal{Q}$, because $\mathcal{Q}$ crosses $\mathcal{P}$ at an intersection vertex I if and only if $\mathcal{P}$ crosses $\mathcal{Q}$ at I.

The labelling of crossing vertices is finally done as described in Section 3 by tracing all components of both polygons $\mathcal{P}$ and $\mathcal{Q}$ once and marking entry and exit points with respect to the other polygon's interior. For the example in Figure 8, this algorithms marks the vertices P_7, P_{13}, Q_2, Q_9 as entry points and the vertices P_9, P_{14}, Q_7, Q_{12} as exit points (see Figure 12 a).

Note that the final labelling stage requires at least one vertex of each polygon component to be non-intersecting, so that the inside/outside test can be executed unambiguously, which may not be the case in some special situations like the one in Figure 13 a. To explain how to overcome this problem, let us assume that some component $\boldsymbol{P}$ of $\mathcal{P}$ consists entirely of intersection vertices after executing the first phase of

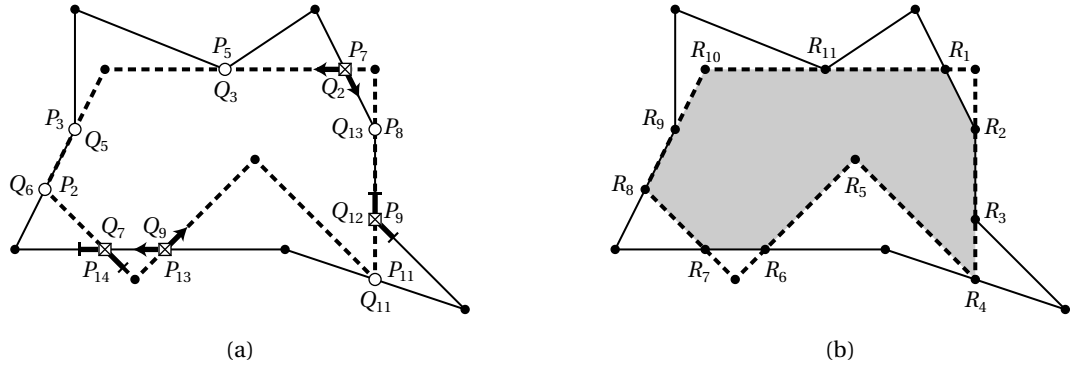


Figure 12: Example from Figure 8 after the second phase (a). The algorithm marks the intersection vertices as crossing (⊠) or bouncing (○) and labels the crossing vertices as entry (⟶) and exit (⟵) points for both polygons. The third phase of the algorithm finally creates the intersection polygon (b).

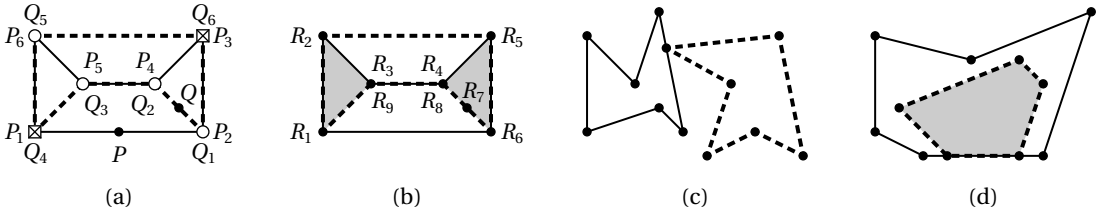


Figure 13: Examples of some special cases. If a polygon component consists of intersection vertices only after phase one (a), then we add the midpoint of the first non-overlapping edge to the polygon, because the entry/exit-classification in phase two requires an initial non-intersection vertex. For the two polygons in (a), the result has one non-simple component $\boldsymbol{R} = [R_1, \ldots, R_9]$ (b). If some polygon component does not contain any crossing vertices after the labelling phase, then it either does not intersect the other polygon (c), or it contains or is contained in a component of the other polygon (d), and then the interior component is added to $\mathcal{R}$.

our algorithm and distinguish two cases. If all edges of $\boldsymbol{P}$ represent common segments, then $\boldsymbol{P}$ and some component $\boldsymbol{Q}$ of $\mathcal{Q}$ enclose the same region and we can simply add $\boldsymbol{P} = \boldsymbol{Q}$ as a component of the intersection polygon $\mathcal{R}$, if and only if either both components are holes or if both are not holes. Otherwise, at least one of the intersection vertices of $\boldsymbol{P}$ is not an *on/on* vertex and thus adjacent to an edge, say $[P_i, P_{i+1}]$, that does not overlap with $\mathcal{Q}$. Hence, we can add the midpoint $P = (P_i + P_{i+1})/2$ as *temporary* vertex to $\boldsymbol{P}$ and use it as the initial non-intersection vertex for the entry/exit-classification.

4.3 Tracing phase

The third phase of the algorithm for creating the intersection polygon $\mathcal{R}$ remains largely unchanged from the original Greiner–Hormann algorithm and is described in Section 3. The only difference is that the generation of each result component starts at a crossing intersection vertex I of $\mathcal{P}$ and that we traverse $\mathcal{P}$ as usual, forward if I is an entry point and backward otherwise, but this time until we reach a vertex of $\mathcal{P}$ with *opposite* entry/exit flag, and likewise after switching over to $\mathcal{Q}$. In the absence of degenerate intersections, this is equivalent to proceeding to the next intersection vertex, but in general we may pass one or more bouncing intersection vertices before switching polygons. For the example in Figure 12 b, we thus start at P_7, traverse $\mathcal{P}$ forward until we get to $P_9 = Q_{12}$, then backward along $\mathcal{Q}$ up to $Q_9 = P_{13}$, and so on.

If some component $\boldsymbol{P}$ of $\mathcal{P}$ does not contain any crossing intersection vertex, then we have encountered one of the special cases shown in Figures 13 c and 13 d, which can be dealt with as follows. Let P be a non-intersection vertex of $\boldsymbol{P}$ or the midpoint of an edge that does not overlap with $\mathcal{Q}$. If and only if P lies inside $\mathcal{Q}$, then so does the entire component $\boldsymbol{P}$ and we add it as a component of $\mathcal{R}$ in this case. The same strategy is applied to all components of $\mathcal{Q}$ that do not contain any crossing intersection vertex, and by examining all possible cases, it is clear that this procedure gives the correct result, even in the case of nested holes.

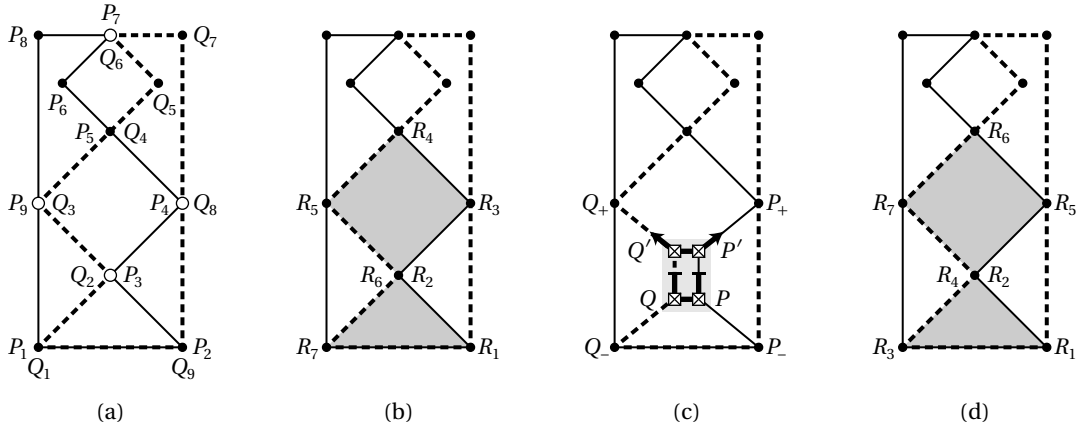


Figure 14: Example of two polygons (a), for which the algorithm without vertex splitting generates a degenerate result with one component $\boldsymbol{R}_1 = [R_1, \ldots, R_7]$ that contains a duplicate vertex $R_2 = R_6$ (b). After splitting the vertex pair $(P, Q) = (P_3, Q_2)$, setting the entry/exit flags, and linking the new vertices P' and Q' (c), the third phase of the algorithm generates the correct result with two simple components $\boldsymbol{R}_1 = [R_1, R_2, R_3]$ and $\boldsymbol{R}_2 = [R_4, R_5, R_6, R_7]$ (d).

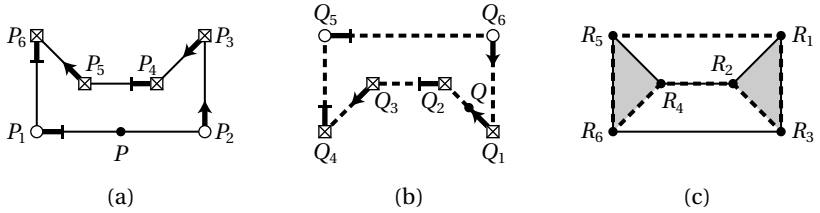


Figure 15: For the example from Figure 13 a, the improved labelling strategy sets the entry/exit flags not only for crossing vertices, but for all endpoints of intersection chains, both for $\mathcal{P}$ (a) and for $\mathcal{Q}$ (b), so that the third phase of the algorithm generates a result polygon with two components $\boldsymbol{R}_1 = [R_1, R_2, R_3]$ and $\boldsymbol{R}_2 = [R_4, R_5, R_6]$ (c).

4.4 Improvements and generalizations

In contrast to the Greiner–Hormann algorithm and as a consequence of the way we handle degenerate intersections, the result generated by the algorithm above may contain three kinds of degeneracies, which are not incorrect per se, but should be resolved in order to make the result as simple as possible.

First, there can be chains of three or more successive, collinear vertices, and all but the first and last vertices of such a chain can be omitted without modifying the correctness of the result. An example is the intersection polygon in Figure 12 b, where the vertices R_3, R_9, and R_{11} should be removed. In general, the vertex R should be removed, if and only if $\mathrm{A}(R_-, R, R_+) = 0$, and this can be done in a post-processing step that visits each vertex of the result once.

Second, it may happen that a vertex appears twice in the result polygon, as shown in Figure 14 b. The polygon should then be split into two parts at such vertex, so as to make all components of $\mathcal{R}$ simple. Notice that this situation can only occur at a bouncing intersection vertex for which the adjacent edges of $\mathcal{P}$ lie inside $\mathcal{Q}$ and vice versa. In order to detect these cases, we extend the final labelling stage of the second phase to mark all bouncing vertices that lie between an entry and an exit point as *split candidates*. For the example in Figure 14 a, the split candidates are the bouncing vertices P_3, P_4, Q_2, and Q_3, but not P_6, P_9, Q_5, and Q_8. After the labelling, we loop through the split candidates of $\mathcal{P}$ and if we encounter a candidate P, whose neighbour Q has been marked as a split candidate for $\mathcal{Q}$, we prepare the split of this vertex pair (P, Q). To this end, we insert a copy P' of P after P into $\mathcal{P}$ and likewise for $\mathcal{Q}$, as shown in Figure 14 c. We then label P and Q as exit and P' and Q' as entry points and mark all four vertices as crossing, so that they can serve as initial vertices for generating the intersection polygons in the third phase. Finally, we need to link them in the correct way. If the local orientation of $\mathcal{P}$ at P and $\mathcal{Q}$ at Q is different, that is,

$$\mathrm{sign}\big(\mathrm{A}(P_-, P, P_+)\big) \neq \mathrm{sign}\big(\mathrm{A}(Q_-, Q, Q_+)\big),$$

as in the example in Figure 14 c, then we keep the link between P and Q and link P' with Q'. Otherwise, we link P with Q' and Q with P'.

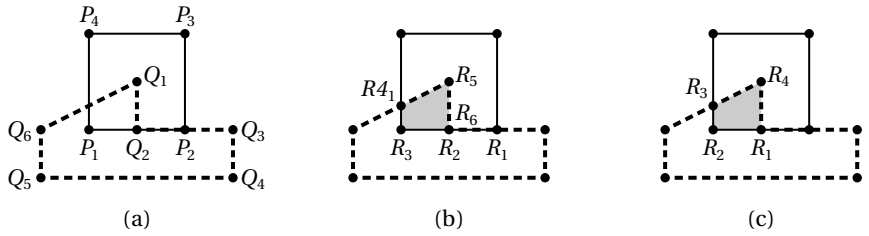


Figure 16: Example of two polygons (a), for which the algorithm generates a degenerate result with glued edges $[R_1, R_2]$ and $[R_6, R_1]$ (b). Both get removed with the improved labelling strategy (c).

Third, the result may contain "glued" edges, like $[R_3, R_4]$ and $[R_8, R_9]$ in Figure 13 b or $[R_1, R_2]$ and $[R_6, R_1]$ in Figure 16 b, which bound an area with no interior and should therefore be removed. While this can also be done in a post-processing step, it is preferable to avoid them upfront. Indeed, we achieve this by two minor modifications of the labelling strategy in the second phase of our algorithm, which are inspired and adapted from the observations in [11]. On the one hand, when classifying the intersection chain $\boldsymbol{I} = (I_1, I_2, \ldots, I_k)$, we mark I_1 and I_k as endpoints of a *delayed crossing* or *delayed bouncing* and the intersection vertices $I_2, \ldots, I_{k-1}$ as bouncing. On the other hand, when tracing $\mathcal{P}$ and $\mathcal{Q}$ for setting the entry/exit flags, we mark the endpoints of a *delayed bouncing* in the same way as regular crossing intersection vertices, while the endpoints of a *delayed crossing* are marked either both as entry or both as exit points. In addition, we mark the first vertex of an exiting *delayed crossing* and the last vertex of an entering *delayed crossing* as crossing. In case of a *delayed bouncing*, we mark both endpoints as crossing, if and only if the adjacent edges of $\mathcal{P}$ lie inside $\mathcal{Q}$ and vice versa. Similar to the vertex splitting above, this requires identifying *crossing candidates* during the traversal and marking matching candidates afterwards. Since the traversal starts at a non-intersection vertex and thus never at an interior vertex of an intersection chain, the two endpoints of an intersection chain are always visited in pairs and it is easy to distinguish the first endpoint from the last.

As the examples in Figures 15 and 16 c show, this improved labelling strategy is able to remove glued edges effectively, and an exhaustive examination of all possible combinations of clockwise- or counterclockwise oriented components of $\mathcal{P}$ and $\mathcal{Q}$, of entering or exiting *delayed crossings*, as well as interior or exterior *delayed bouncings*, reveals that this strategy handles all cases correctly; see [11] for details.

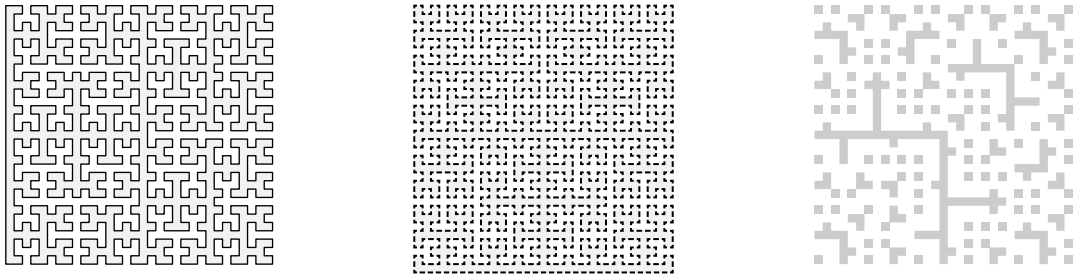
We finally note that, just like the original Greiner–Hormann algorithm [5], our extended version can also compute the union of $\mathcal{P}$ and $\mathcal{Q}$ after some minor modifications. The key change is to reverse the traversal directions during the tracing phase and to travel forward from exit to entry points along the polygons and backward from entry to exit points. Moreover, entire components must be added to $\mathcal{R}$ if they lie outside instead of inside the other polygon, and the rules for splitting vertices and the labelling of endpoints of *delayed crossing* and *delayed bouncings* must be reversed. Similar changes can be made for determining the differences $\mathcal{P} \setminus \mathcal{Q}$ and $\mathcal{Q} \setminus \mathcal{P}$.

5 Examples

We implemented the algorithm in `C++` and tested it extensively for various input polygons. The code and all examples are available on the second author's webpage at `https://www.inf.usi.ch/hormann/polyclip/`.

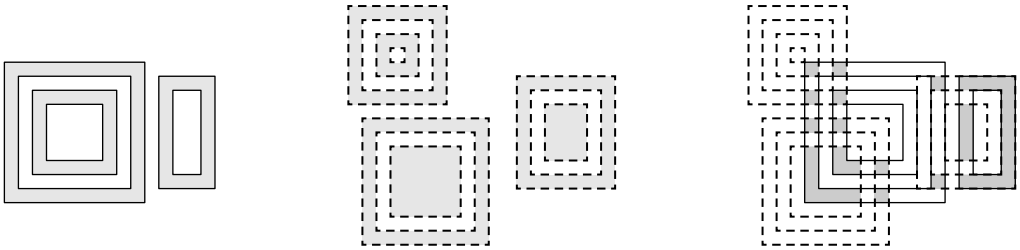
The first example in Figure 17 is a real stress test for degenerate intersections, as both input polygons, with 820 vertices each, interpolate all nodes of a regular 32×32 grid. The first phase of the algorithm finds 14 non-degenerate and 1010 degenerate intersections (176 T- and 432 V-intersections, as well as 208 T- and 194 V-overlaps) and adds 206 vertices to both polygons. The second phase detects 42 bouncing intersection vertices and 396 delayed crossings, and splits 21 bouncing vertex pairs. The third phase creates 116 simple polygons with 804 vertices, and 164 of these are removed by the post-processing step that eliminates collinear vertices.

The second example in Figure 18 illustrates that the algorithm also handles complex input polygons with multiple and nested components, with the interior defined by the even-odd rule [4]. There are 46 non-degenerate and 21 degenerate intersections in this example, with the latter corresponding to 4 bouncing intersection vertices, 2 delayed crossings, and 5 delayed bounces. No bouncing vertex pairs need to be split, and after removing 3 collinear vertices, the result consists of 14 simple polygons with 64 vertices.



(a) (b) (c)

Figure 17: The intersection of a closed fifth-order Hilbert curve (a) with a rotated copy of itself (b), aligned at the top right, gives 116 simple polygons (c).



(a) (b) (c)

Figure 18: The algorithm also handles complex input polygons with multiple and nested components (a,b) and computes the intersection correctly (c).

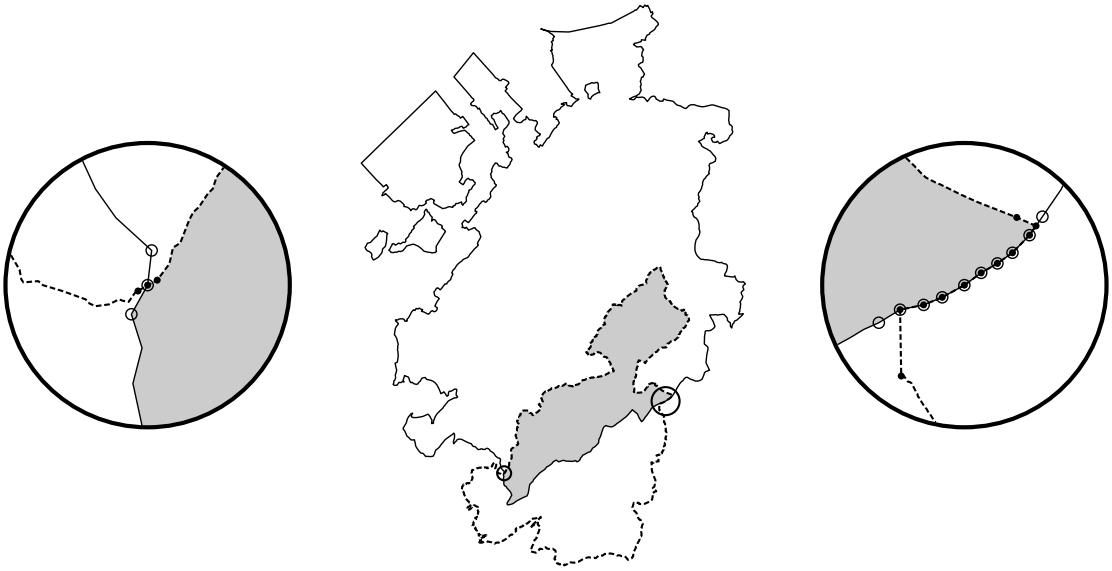


Figure 19: Intersection (shaded) of the boundary of the *Regional Nature Park Gruyère Pays-d'Enhaut* (dashed lines and dotted vertices) with the boundary of the *Canton of Fribourg* (solid lines and circled vertices), with close-ups to the regions where both boundaries intersect.[1]

The last two examples testify to the need of being able to handle degenerate cases in real-world applications. In Figure 19 we show the polygons representing the boundaries of a canton (5 components, 1322 vertices) and a nature park (1 component, 2602 vertices) in Switzerland. The close-ups zoom to the crossing and the delayed crossing (right) found by the algorithm. Comparing the area of the intersection result (1240 vertices) with the area of the nature park reveals that 46.6% of the park belong to this canton.

[1] Data used in this example is from `www.openstreetmap.org`, made available under the *Open Database License* (ODbL).

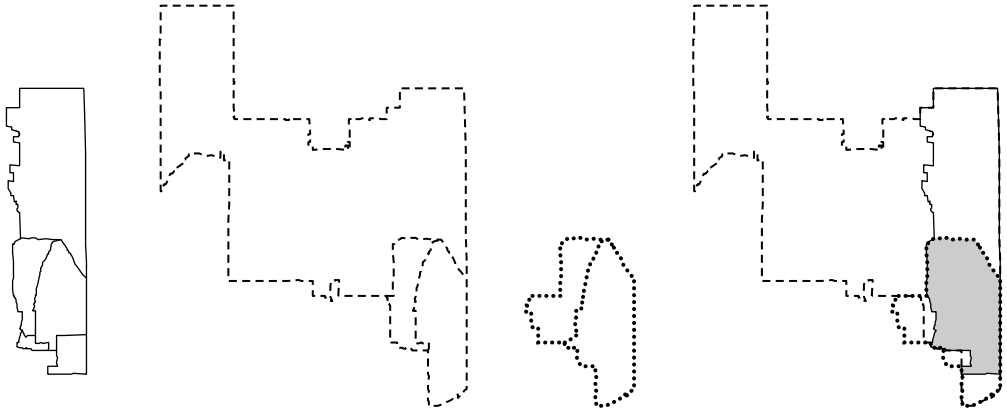


Figure 20: To find the common region of five elementary school districts (a), three middle school districts (b), and two high-school districts (c), we first compute their respective unions and finally their intersection (d).

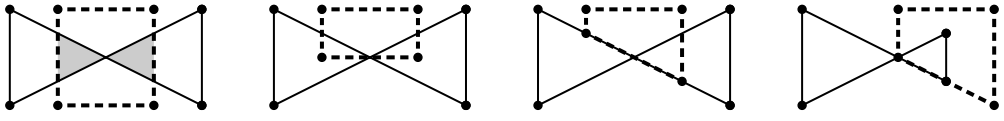
In the final example in Figure 20, we consider polygons representing US school districts in the Albuquerque region, ranked by school rating. We first compute the union of the polygons for the 5 best-ranked elementary schools (which requires to run the algorithm four times, adding one polygon at each run) and likewise for the 3 best-ranked middle schools and the 2 best-ranked high schools, resulting in the polygons $\mathcal{E}$, $\mathcal{M}$, and $\mathcal{H}$. We then compute the intersection of $\mathcal{E}$ with $\mathcal{M}$ and further intersect the result with $\mathcal{H}$. This finally gives the shaded polygon in Figure 20 d, which represents the neighbourhood with access to top schools on all three levels of schooling.

6 Discussion and conclusions

Clipping planar polygons is central to several fields, and the need for a general algorithm capable of clipping convex and concave polygons with multiple components and holes was pointed out by Weiler and Atherton [17]. Their algorithm was the first to have this feature, and is akin to our work in that it consists of an intersection and a tracing phase that are basically the same as ours. The Weiler–Atherton algorithm gets by without a labelling phase, since it assumes the vertices of all polygon components to be ordered consistently, namely clockwise for exterior boundaries and counter-clockwise for holes. By adding the labelling phase, Greiner and Hormann [5] manage to avoid this restriction on the vertex order and to generalize the Weiler–Atherton algorithm so that it also handles self-intersecting polygons correctly. However, both algorithms cannot deal with degenerate intersection cases, which is a severe limitation in many applications.

Weiler's polygon comparison algorithm [16] overcomes this drawback, albeit at the expense of using a more complicated graph data structure. Instead, we show that degenerate intersection cases can also be dealt with effectively by carefully refining the labelling phase of the Greiner–Hormann algorithm. Our new labelling phase (Sections 4.2 and 4.4) is efficient, since it relies on strictly local operations and on detecting and distinguishing a small number of cases. In fact, the running time of this phase is $O(k)$, where k is the number of intersections between $\mathcal{P}$ and $\mathcal{Q}$, and according to our experience it takes only about twice as long as the original labelling phase of the Greiner–Hormann algorithm. The only global information needed for labelling all entry/exit flags correctly at the end of this phase is the inside/outside test that is applied to one non-intersection vertex for each polygon component and typically requires $O(n)$ operations, where n is the number of vertices of the other polygon. Note that the algorithm of Kim and Kim [7], which also extends the Greiner–Hormann algorithm to handle degenerate intersections, requires carrying out two inside/outside tests for each intersection vertex or intersection chain, resulting in an inferior $O(kn)$ time complexity.

Overall, our algorithm is only marginally slower than the original Greiner–Hormann algorithm, because the running time is dominated by the intersection phase, which usually takes more than 80% of the time, so that the small overhead induced by the new labelling phase is negligible. As proposed by Greiner and Hormann [5], we adopt the brute force approach for the intersection phase and find the k intersections of $\mathcal{P}$ and $\mathcal{Q}$ by simply testing all n edges of $\mathcal{P}$ against all m edges of $\mathcal{Q}$, which obviously requires $O(nm)$ operations, and is the best one can do in the worst case, when $k \in O(nm)$. However, if the number of intersections is



(a) (b) (c) (d)

Figure 21: Our algorithms handles self-intersecting polygons correctly, as long as the self-intersection does not lie on the other polygon (a), but may fail otherwise (b, c, d).

small, then it is more efficient to compute them with a plane sweep approach in $O((n+m+k)\log(n+m))$ time [2]. The latter is done by Vatti's algorithm [14], the algorithm of Martínez, Rueda, and Feito [10], and its successor [9], which are also able to deal with degenerate intersections and are reportedly faster than the Greiner–Hormann algorithm if $k \in O(n+m)$ [9, 10].

Our experiments confirmed these timings, and also those in [5], which show that both the Greiner–Hormann and our algorithm outperform the plane-sweep-based approaches for completely random polygons with many self-intersections, simply because the latter do not have to be computed. Moreover, our algorithm is the fastest in case of moderately sized polygons ($mn \leq 1\,000\,000$, $k \leq 10\,000$), most probably because of the extremely efficient labelling and tracing phases, which both take less than 1 ms in this case. It is very likely that the algorithm can further be sped up by employing the plane sweep approach in the intersection phase, but it remains future work to verify this conjecture. Further improvement, also in terms of memory efficiency, is probably possible by adapting the idea of Liu et al. [8], who suggest to maintain a single doubly-linked list of intersections, instead of inserting them into $\mathcal{P}$ and $\mathcal{Q}$.

The only limitation of our algorithm are degenerate intersections involving self-intersection points. In fact, if a self-intersection of one polygon lies on an edge or coincides with a vertex of the other polygon, then the algorithm may fail (see Figure 21). However, this problem can be avoided by resolving self-intersections in a preprocessing step, for example by splitting $\mathcal{P}$ in Figure 21 into two triangles.

It remains future work to find a more elegant solution to the aforementioned limitation and to extend the ideas presented in this paper to the 3D setting, so that they can be used for Boolean operations in solid modeling, model repair, and other geometry processing applications [15].

References

[1] M. K. Agoston. Clipping. In *Computer Graphics and Geometric Modeling: Implementation and Algorithms*, chapter 3, pages 69–110. Springer, London, 2005.

[2] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*, chapter 2, pages 19–43. Springer, Berlin, 3rd edition, 2008.

[3] P. E. Farrell, M. D. Piggott, C. C. Pain, G. J. Gorman, and C. R. Wilson. Conservative interpolation between unstructured meshes via supermesh construction. *Computer Methods in Applied Mechanics and Engineering*, 198(33–36):2632–2642, July 2009.

[4] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley Systems Programming Series. Addison-Wesley, Reading, 2nd edition, 1990.

[5] G. Greiner and K. Hormann. Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics*, 17(2):71–83, Apr. 1998.

[6] K. Hormann and A. Agathos. The point in polygon problem for arbitrary polygons. *Computational Geometry: Theory and Applications*, 20(3):131–144, Nov. 2001.

[7] D. H. Kim and M.-J. Kim. An extension of polygon clipping to resolve degenerate cases. *Computer-Aided Design and Applications*, 3(1–4):447–456, 2006.

[8] Y. K. Liu, X. Q. Wang, S. Z. Bao, M. Gomboši, and B. Žalik. An algorithm for polygon clipping, and for determining polygon intersections and unions. *Computers & Geosciences*, 33(5):589–598, May 2007.

[9] F. Martínez, C. Ogayar, J. R. Jiménez, and A. J. Rueda. A simple algorithm for Boolean operations on polygons. *Advances in Engineering Software*, 64:11–19, Oct. 2013.

[10] F. Martínez, A. J. Rueda, and F. R. Feito. A new algorithm for computing Boolean operations on polygons. *Computers & Geosciences*, 35(6):1177–1185, June 2009.

[11] R. T. Popa, E.-C. Mladin, E. Petrescu, and T. Prisecaru. A simple en,ex marking rule for degenerate intersection points in 2D polygon clipping. arXiv:1709.00184 [cs.CG], 2017. `https://arxiv.org/abs/1709.00184`.

[12] A. Schettino. Polygon intersections in spherical topology: Applications to plate tectonics. *Computers & Geosciences*, 25(1):61–69, Feb. 1999.

[13] L. J. Simonson. Industrial strength polygon clipping: A novel algorithm with applications in VLSI CAD. *Computer-Aided Design*, 42(12):1189–1196, Dec. 2010.

[14] B. R. Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63, July 1992.

[15] C. C. L. Wang and D. Manocha. Efficient boundary extraction of BSP solids based on clipping operations. *IEEE Transactions on Visualization and Computer Graphics*, 19(1):16–29, Jan. 2013.

[16] K. Weiler. Polygon comparison using a graph representation. *Computer Graphics*, 14(3):10–18, July 1980. Proceedings of SIGGRAPH.

[17] K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. *Computer Graphics*, 11(2):214–222, July 1977. Proceedings of SIGGRAPH.