MSc thesis in Geomatics

Snap rounding polygons with a triangulation

Fengyan Zhang

June 2023

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

The work in this thesis was carried out in the:

3D geoinformation group
Delft University of Technology

Supervisors: Dr. Hugo Ledoux
Dr. Ken Arroyo Ohori
Co-reader: Dr.ir. BM Meijers

Abstract

Geometric algorithms are usually designed based on the assumption of using arbitrary-precision representations. However, in modern computer systems floating-point arithmetic and finite-precision approximation are often used as implementing exact computations would require large computational resources and would be rather slow in the applications. Snap rounding (SR) is a widely used technique to transform representations of infinite precision into fixed-precision formats. It slightly modifies the original geometry of the input in order to obtain a better organized geometric object arrangement and avoid the issues caused by applying floating-point arithmetic in geometric algorithms. The existing implementations of SR primarily focus on processing sets of line segments, while in practice polygons are of great significance as well yet has not received adequate attention so far.

In this thesis, a new method is proposed to perform SR on 2-dimensional polygons. Firstly the input polygons are embedded into a triangulation. Subsequently the triangulation is tagged with the ID information of polygons and the polygon boundaries are extracted and stored in a container. The boundaries and the triangulation are then dynamically processed according to the identification of close polygonal vertices and close vertex and boundaries (under a predefined tolerance). After having snapped all the close elements the rounded polygons will be reconstructed from the processed boundaries. The testing results show the proposed method is capable of eliminating small gaps between elements (i.e. vertices and boundaries) and is adaptable to various input. The topological and geometrical properties of polygons are preserved as much as possible. Finally, an overview of the limitations is provided, along with potential directions for future research.

Acknowledgements

I would like to express my sincere gratitude to all those who have helped me during the thesis. First of all, I am deeply grateful to Hugo Ledoux and Gustavo Adolfo Ken Arroyo Ohori for their invaluable guidance and support throughout my thesis journey. They are not only my mentors for this specific thesis but also my mentors who help me a lot regarding various courses and programming challenges. Their prior work has laid the foundation for this thesis, and I feel fortunate to have had the opportunity to build upon their expertise. Throughout the thesis completion process, I encountered numerous obstacles, ranging from programming and development issues to difficulties in academic writing. In these moments of uncertainty, Hugo and Ken demonstrated unwavering enthusiasm, generosity, and patience in addressing my questions. Without their support, this thesis would have been significantly more challenging. Their contributions have played a vital role in the progression of this thesis.

Next, I would like to express my gratitude to Giorgio Agugiaro and Martijn Meijers for their participation in my thesis and providing valuable feedback. Their sparkling wisdom and valuable advice have contributed a lot with regard to improving my thesis. I am very grateful for their professionalism and the time they have put in for me.

Finally, I would like to extend my deepest gratitude to my family and my girlfriend. During moments of pressure and confusion, it is their encouragement and patience that helped me maintain the courage to continue moving forward. I am truly grateful. It is their love that keeps me pushing myself towards higher and better directions. Thank you for the care and companionship during the this thesis.

Thanks to everyone. I love you all.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

1 Introduction

1.1 Research Motivation

Geometric objects are commonly described in a continuous space, e.g. the Euclidean space $\mathbb{R}^2$ and $\mathbb{R}^3$, this also introduces the infinite-precision (or arbitrary-precision) representations of geometric objects [Goodrich et al., 1997]. The infinite-precision usually stands for the exact computations, which are commonly used when we deliberate and design the logic of a geometric algorithm. For instance, finding the intersection of two line segments. This seems to be a quite straightforward task, yet it is actually not the case inside a computer. Various issues will arise when it comes to the implementation, especially when robustness and reliability are desired [Halperin and Packer, 2002]. The major problem is that, a computational model is usually proven to be correct under the assumption of using exact computations rather than floating-point arithmetic. In practice, however, the hardware of modern computers does use floating-point arithmetic to replace the exact arithmetic. For some applications this can deliver relatively good results while in scenarios where high precision is required, floating-point arithmetic can lead to severe errors such as program crashing, infinite loop, or yielding wrong results [Schirra, 1998].

One prevalent solution is using exact computations provided by libraries like Computational Geometry Algorithms Library (CGAL) [Fabri and Pion, 2009]. Exact computations indeed can produce reliable results while it usually consumes large resources and will slow down the application. Another substitution is to use finite-precision approximation to better organize the arrangement of geometric objects and to overcome the possible degenerate problems caused by precision issues. SR is such a method to convert an arrangement of objects (e.g. line segments) into a fixed-precision representation. It slightly modifies the original geometry of the input data, which is also known as a perturbation [Raab, 1999], to make the geometric objects better organized.

In recent researches, various approaches have been proposed and implemented to snap rounding a set of line segments, such as *iterated snap rounding* [Halperin and Packer, 2002], *iterated snap rounding with bounded drift* [Packer, 2006], *snap rounding with restore* [Belussi et al., 2016], etc. They all have achieved promising results and have been applied in the industry. One can arguably say that the algorithms towards line segments have attained a state of maturity. However, in 2-dimensional space, not only should the attention be given to line segments, but equal consideration should also be given to polygons, especially in the field of Geographical Information System (GIS), as polygons are also of great importance in the geometric computations and analyses. For instance, in the spatial analysis, if the relationships of polygons are incorrectly interpreted due to small errors (e.g. small gaps), then the relevant calculations (such as area calculation) may be underestimated or overestimated. This is undesirable in most applications and should be avoided whenever possible.

Pertaining to this, it has been shown that a constrained triangulation (CT) can be used as a supporting data structure to efficiently perform operations related to polygons. Algorithms

have been developed with the objective of validating and correcting polygonal geometries, such as fixing invalidities of individual polygons [Ledoux et al., 2012] and repairing a planar partition [Ohori et al., 2012]. Generally speaking, utilizing a CT provides a good basis for implementing various algorithms towards polygons with ease and efficiency.

Based on the previous researches, in order to investigate the possibility of snap rounding polygons in $\mathbb{R}^2$, a new approach is intended to be proposed and developed in this thesis. Similarly, it will utilize a CT as an auxiliary data structure. An arrangement of 2-dimensional polygons will be used as input and the output will be its rounded counterpart. The testing and benchmarking results will be presented as well.

1.2 Research Objectives

Summarizing the content above, we define the *snap rounding of polygons* as such:

- *snap rounding polygonal vertices*: if the distance between two vertices is less than a predefined threshold, then the two vertices are considered the same and should be represented with one vertex.
- *snap rounding polygonal vertex and boundaries*: if the distance between a vertex and a boundary is under a certain threshold, then the vertex should be touching with the boundary.

Based on this, the main research question of this thesis is: *How can a constrained triangulation (CT) be used as a supporting data structure to robustly perform snap rounding on polygons in 2-dimensional space?*

In order to answer the question, the following sub-tasks will be gradually studied:

- How to integrate a CT with the input polygons (possibly with interior holes)?
- Is it better to have a certain type of a CT such as a constrained Delaunay triangulation (CDT) or is any type of a CT fine?
- How to link the triangulation with the original polygons? If the triangulation is modified, how to update the changes to the polygons?
- Polygons usually have attributes attached (e.g. polygon id, area of a polygon), how to preserve them in a proper way during the SR process?
- Given a certain threshold, there may exist several SR cases (e.g. multiple polygonal vertices and polygonal vertex and boundaries), what is the most reasonable order when SR is being performed?
- How to measure and evaluate the distortions of the polygons before and after SR?

To achieve the goal, an algorithm of automatically performing snap rounding on polygons will be developed. It will use a CT or possibly its variation, a CDT as a basic supporting data structure. Although robustness is our first priority, efficiency should also be weighted and evaluated. Moreover, external libraries will be used, such as Geospatial Data Abstraction Library (GDAL) [GDAL/OGR contributors, 2022] for reading and writing files and CGAL [Fabri and Pion, 2009] for using robust geometric algorithms including building a CT or a CDT [Boissonnat et al., 2000].

1.3 Challenges and Scope

Integrating a CT and polygons has already been studied and implemented in [Ledoux et al., 2012] and [Ohori et al., 2012]. The main challenge of this thesis is how to keep track of the information of polygons while the triangulation is being modified. In other words, the polygons need to be updated without losing any information (i.e. polygon ID) as the changes on the triangulation are made. [Ohori et al., 2012] proposed and implemented a method to label the triangulation in accordance with the id of input polygons. However, in this thesis the triangulation will be dynamically changed and will invalidate some triangles that have already been labelled, which will cause the lack of information.

It is difficult to identify the invalidated triangles, for the triangulation can have complex and irregular structures. Therefore, in this thesis a new method needs to be developed to track the information of polygons while their geometries are being modified in the triangulation.

The thesis will focus on the polygons, thus the input and output will all be polygons. The aim is to solve the problems mentioned in Section 1.2 via performing *snap rounding on polygons*. Other topological errors amongst polygons will not be considered as main problems to tackle, although some cases such as overlaps and self-intersections can be handled by snap rounding to some extent.

1.4 Thesis Outline

The outline of this thesis is shown as follows:

- **Chapter 2** introduces the background and basic information pertaining to this thesis, including the well-known floating-point arithmetic, the origin and development of snap rounding and the definition of a CDT. Furthermore, in this chapter different algorithms that perform snap rounding operations on a set of line segments are reviewed and discussed.
- **Chapter 3** presents the whole pipeline of the proposed method. It mainly includes polygon embeddings, tagging of the triangulation, snap rounding polygonal elements (vertices and boundaries) and polygon reconstructions. Figures and pseudo-code are frequently used to elaborate the procedures.
- **Chapter 4** mainly describes the implementation details, including data structures to be used, the developed prototype and related engineering decisions.
- **Chapter 5** presents relevant results and analyses of some datasets. The evaluation method is also explained in this chapter.
- **Chapter 6** gives the conclusion of this thesis and possible directions of the future work.

2 Related work

2.1 Floating-point Arithmetic

In the introduction, it has been described that modern computers are using floating-point arithmetic to replace the exact arithmetic in common practice. Furthermore, floating-point is widely utilized in various ways regarding computer systems. For instance, a majority of popular programming languages (e.g. C++, Python, Java, etc.) have their own internal floating-point data types and almost all of the modern computers have floating-point accelerators [Goldberg, 1991]. Note however, floating-point has its own limitations. The most important, and perhaps the most talked about aspect is the rounding error. In computer, real numbers can not be exactly represented with infinite number of bits thus must be replaced with finite precision (fixed precision). In practice, computations with real numbers will also yield results that typically can not be precisely represented with a given number of bits. Therefore, rounding operations are required so as to reasonably express the real numbers via a finite manner but meanwhile some errors will also be introduced.

Regarding the specific representation of floating-point, it typically has a flexible form shown below:

$$\pm d.dd\ldots d \times \beta^e$$

where $d.dd\ldots d$ is called the *significand*, which was firstly used by [Forsythe and Moler, 1967] to replace the conventional term *mantissa*. It includes p digits (p also represents the precision, the more digits after the decimal point are, the higher the precision is). β is called the *base* which is commonly postulated to be an even number, with e being its exponent.

In order to gain a better understanding of the representation, let us take a look at an example. Suppose we have $\beta = 10$ and $p = 3$, then the number 0.1 can be expressed as 1.00×10^{-1}. But if we make the β have value 2 and p have value 24, the value 0.1 can no longer be precisely represented, it can now only be approximated as: $1.10011001100110011001101 \times 2^{-4}$. In fact this is a well-known example in the field to illustrate one of the consequences using the floating-point arithmetic: not all decimal values (actually there are many in common practice) can be representable [Goldberg, 1991]. It is noteworthy that there are mainly two different kinds of IEEE standards that are used with regard to the floating-point. The most commonly used one is IEEE 754 [198, 1985]. It regulates that β should be having value 2, p should be equal to 24 for single precision and 53 for double precision. In practice, the double precision utilizes one bit to store the sign of the number, 52 bits to store the significand and 11 to store the exponent. This is used by the ESRI shapefiles [Ohori et al., 2012]. The other standard is the IEEE 854, which allows either β having the value 2 or 10 [198, 1987]. Additionally it does not have a rigid regulation of the value of p, instead it specifies the usable values of p for single and double precision respectively [Goldberg, 1991].

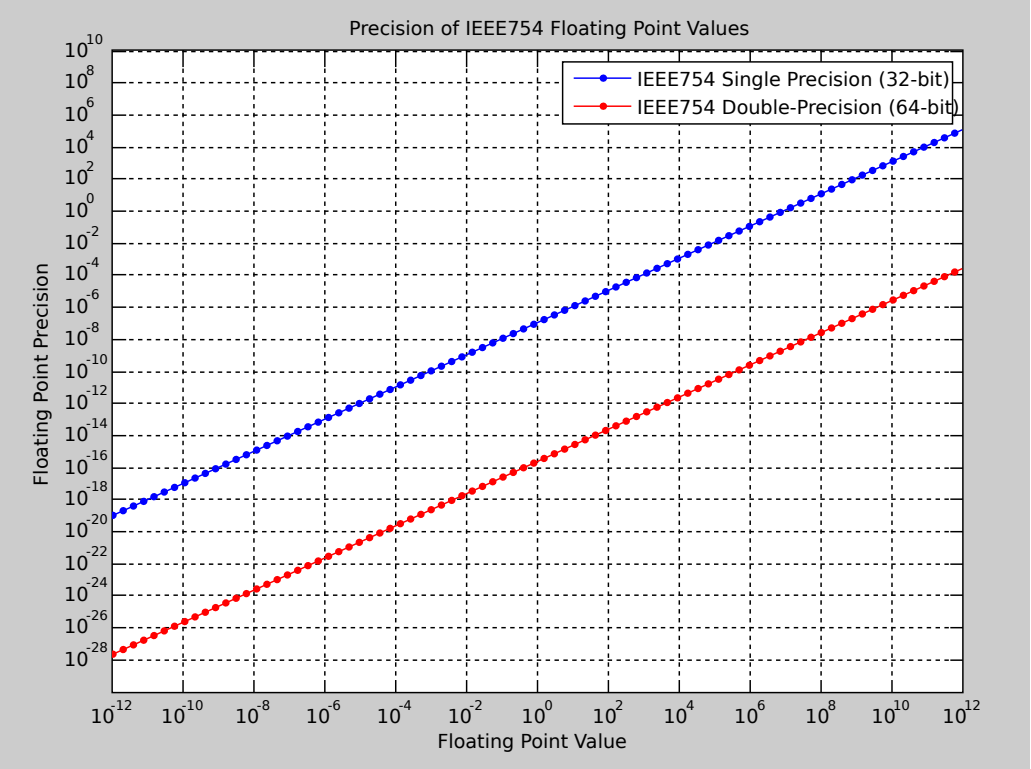


Figure 2.1: Precision of binary32 and binary64 in the range 10^{-12} to 10^{12}. Source: IEEE 754. (2023, April 11). In Wikipedia. https://en.wikipedia.org/wiki/IEEE_754. Author: By Vectorization: Alhadis - Own work based on: IEEE754.png by Ghennessey, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=87066073

2.2 Issues of Using Floating-point Arithmetic

Albeit floating-point arithmetic is nowadays widely accepted and utilized by most of the implementors, it does have some natural consequences that would probably cause problems in the field of geometric computations.

In the first place, the floating-point precision (density of numbers) is at highest level when the value is close to zero, and the precision would gradually decline as the value gets farther from it [Ohori et al., 2012]. This is depicted in Figure 2.1, where the precision reaches its highest when the value is very close to zero (10^{-12}). If the magnitude of a floating-point is relatively big, e.g. 10^{12}, the corresponding precision will be relatively low, e.g. 10^4 to 10^6 for single precision (32-bit) and 10^{-4} to 10^{-2} for double precision (64-bit). To better illustrate this, assume there exists a vertex p, it is firstly projected using a Coordinate Reference System (CRS) which produces large coordinates (let us call it CRS_{LARGE}), an example of possible projected coordinates (XY) could be:

$$(85739.0554312, 446961.2992009)$$

where a range of values from about 85,000 to 450,000 need to be represented, this would require a comparatively large *exponent* to be used. Consider that the total number of available bits used for storing the *significand* and the *exponent* is fixed (although this could vary for different platforms) and the number of bits used for the *significand* determines the precision, a large exponent will consume more bits hence leave fewer bits available for *significand*, resulting in a lower precision.

Then apply another CRS which produces small coordinates (CRS_{SMALL}), an example of possible projected coordinates (XY) could be:

$$(8.57390526216, 4.469619220309)$$

where a range of values from about 8 to 5 need to be stored, this requires a smaller *exponent* to be used thus leave more bits available for the *significand*, resulting in a higher precision.

Observing from above, one can conclude that the magnitude of a number will affect the precision of its corresponding floating-point representation stored in the computer. The smaller the magnitude is, the higher the precision would be, and vice versa.

In the context of geometric field, the precision issues (stated above) are often caused by data manipulation and exchange. They are rather common amongst different users with different aims. For instance, for one specific dataset, various CRS may be used to fit different needs. As mentioned above, the utilization of diverse CRS can lead to significant disparities in coordinate values, potentially encompassing substantial changes in the magnitude of the numbers. Since the precision is not the same regarding different magnitude of values, precision loss could happen, which would cause severe consequences in some cases. See Figure 2.2 for an illustration. Figure 2.2a shows a valid polygon $\mathcal{P}$, with a vertex d being very close to the boundary ab. The coordinates of all vertices are stored as floating-point numbers. Due to the possible precision issues and rounding errors, the geometric characteristics of $\mathcal{P}$ is not always guaranteed. More precisely, the geometric relationship between vertices could change as precision changes. As is shown in Figure 2.2b and Figure 2.2c, the vertex d might shift onto the boundary ab (creating two adjacent polygons) or shift to the left side of ab (creating invalid self-intersected polygon) with different precisions of different magnitude of coordinates. Such unintended changes could cause unreliable behaviours or even crashes of geometric algorithms (e.g. *segfault*). Additionally, operations like moving a polygon or changing the representation (e.g. converting from a 64-bit to a 32-bit one) could also cause problems due to the precision issues [Ohori et al., 2012].

Another limitation of floating-point arithmetic is that in practice there can be many numbers that are not representable. A good example (0.1) has already been discussed in the previous section. Numbers like 0.1 can only be approximated with the closest floating-point number in the computer memory [Goldberg, 1991]. In the field of geometric computations, this usually indicates that vertices would possibly be stored to a slightly different position comparing to its location in the input [Ohori et al., 2012], which also has the possibility to cause invalidities illustrated in Figure 2.2.

Given these considerations, the geometric algorithms are not always considered safe to use in conjunction with the floating-point arithmetic, especially for those which are highly sensitive to the precisions. Problems can arise in various operations such as the *point-in-polygon* query, *orientation* test and *convex hull* operations [Kettner et al., 2008]. See Figure 2.3 for an example of different Delaunay triangulations generated by exact and inexact (floating-point) arithmetic. The Delaunay triangulation on the left is produced by utilizing the exact

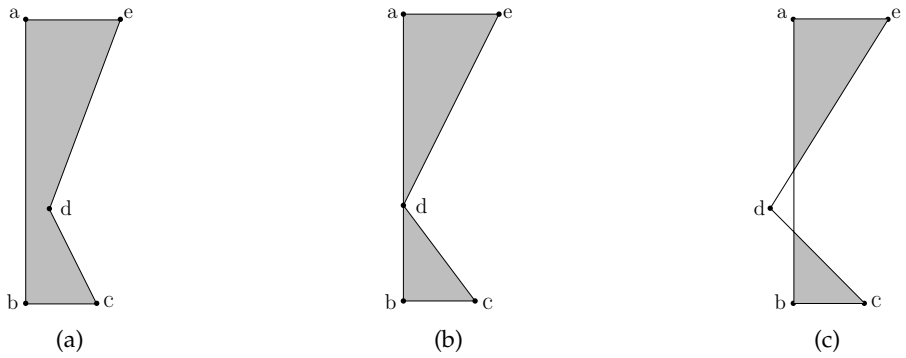


Figure 2.2: Invalid polygons caused by precision issues. **(a)** a valid polygon. **(b)** an invalid polygon which actually consists of two adjacent polygons. **(c)** an invalid polygon with self-intersections.

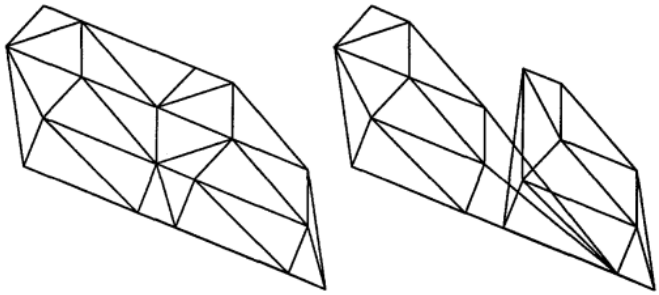
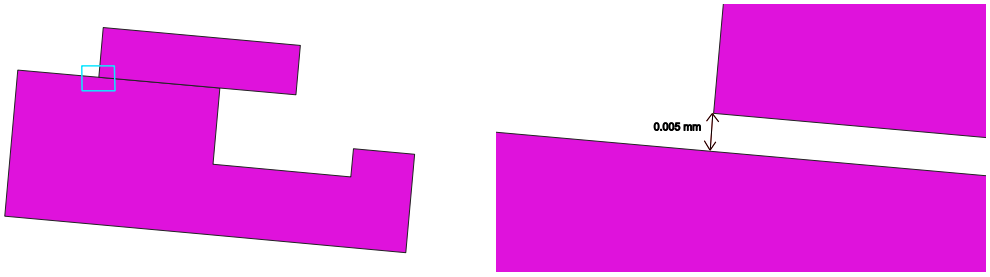


Figure 2.3: Using floating-point arithmetic to construact a Delaunay triangulation. Left: A valid Delaunay triangulation produced by exact arithmetic. Right: An invalid Delaunay triangulation due to rounding errors of floating-point arithmetic. [Shewchuk, 1996]

arithmetic while the right one is generated with floating-point arithmetic. As is shown in the right figure, the triangulation is faulty due to the failure of *orientation* test and *point-in-circle* query. These two predicates are highly sensitive to the precision, thus rounding errors of floating-point arithmetic will make the predicates unstable or even produce incorrect results. More details can be found in [Shewchuk, 1996].

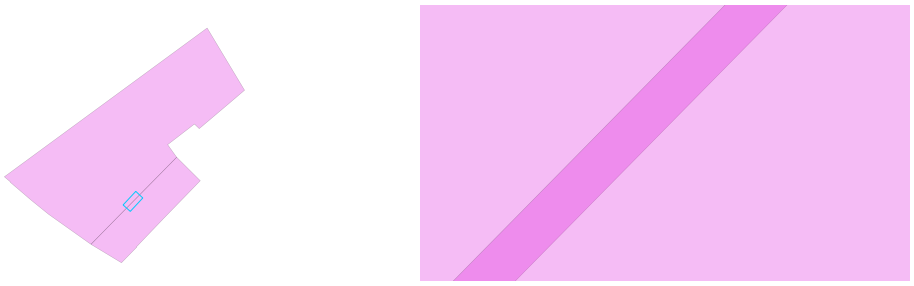
2.3 Limitations of the Data

Excluding the errors caused by floating-point arithmetic, the real-world datasets also contain other types of errors due to several reasons, such as the limitations in the data acquisition stage [Sehra et al., 2016], the complex structures of the real world objects (especially in terms of buildings), automatic and manual intervention of the data creation, manipulation and exchange [Gold et al., 1996; Ohori et al., 2012], etc. Take OpenStreetMap (OSM) [Bennett, 2010] for an example. Data are collected in a crowd-sourcing manner [Haklay, 2010] hence there are possibly several issues such as geographical and topological inconsistency, structural topological consistency, and semantic information that need to be taken into consideration [Sehra et al., 2020]. In this thesis, the main interest lies in the geographical and



(a) (b)

Figure 2.4: Two close polygons. (a) Two close polygons with a scale of 1 : 50. (b) Two close polygons (enlarged) with a scale of 50 : 1. The distance between two boundaries on the map is $0.005mm$.



(a) two overlapping polygons with a scale of 1 : 200

(b) two overlapping polygons with a scale of 50 : 1

Figure 2.5: Two overlapping polygons

topological aspect of the datasets. It has been a consensus amongst specialists in the GIS community that users should be aware of the quality of the input datasets and should avoid easily considering the data given to them is already valid [Ohori et al., 2012].

Figure 2.4 depicts an example of two close polygons. They are extracted from the Netherlands dataset downloaded from GeoFarbik [GeoFabrik, 2023], and then projected using Dutch projected coordinate system (EPSG:28992 - Amersfoort / RD New). Figure 2.4a shows an arrangement of two close polygons from the perspective of most users. The view seems to indicate that these two polygons are adjacent, which means that they share a common boundary and preferably no gaps exist between the two polygons. However, as is shown in Figure 2.4b, if the view is enlarged to a certain level, a discovery can be found that these two polygons are not actually touching (this is quite frequent in the datasets). The small gap (in Figure 2.4b) can cause the failure of an algorithm such as calculating the length of the common boundary of the two polygons [Ohori et al., 2012]. The tricky part is that these close polygons are not obvious to identify with a normal scale hence they can be easily neglected [Laurini and Milleret-Raffort, 1994]. One may only notice the issue when a geometric computation yields errors or warnings. Additionally, the actual value of the small distance on the map in Figure 2.4b was measured via QGIS [QGIS Development Team, 2009]. The measuring result was merely 0.005 millimeter. Such small distance clearly should not be considered as the real gap between the buildings.

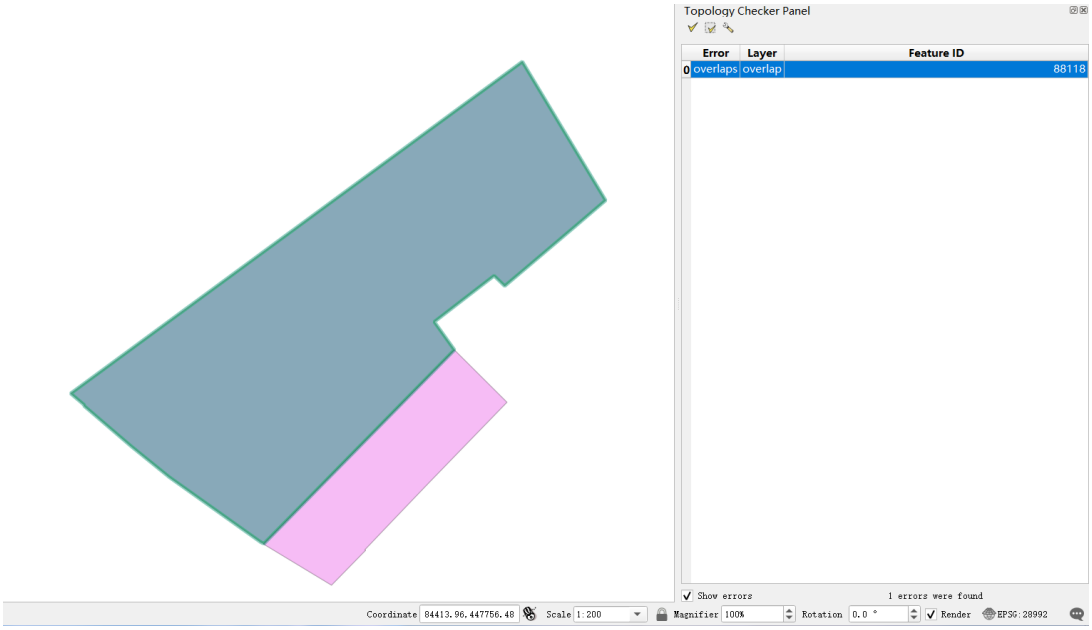


Figure 2.6: Use *topology checker plug-in* to recognize the overlapping polygon(s) in QGIS [QGIS Development Team, 2009]

Another problem that frequently occurs is the small overlapping area. See the example shown in Figure 2.5a. Similarly, the polygons that overlap with each other can be barely observed with a normal scale (e.g. 1 : 200). However, their boundaries actually coincide with each other (see Figure 2.5b with a scale of 50 : 1) and thus will cause issues when one try to perform geometric calculations. For instance, the total area of the two polygons can be correctly computed without overlapping part, yet the computation result will be overestimated with the overlap involved [Ohori et al., 2012]. An additional supporting case is that using the the 9-intersection model [Ubeda and Egenhofer, 1997] to determine the topological relationship of these two polygons. The two polygons are considered as *overlapping* rather than *touching* (which they should be, for the two buildings can not actually overlap in reality).

In the practice, the aforementioned two problems commonly exist and relatively complicated to deal with. The main challenge is that these issues are not obvious and straightforward in accordance with user observations, and they usually can not be automatically corrected by GIS softwares and tools with ease. In QGIS [QGIS Development Team, 2009] for instance, the *topology checker plug-in* tool can recognize the overlapping part, see Figure 2.6 and Figure 2.7 for an example. Users can manually set the rule *must not overlap* to allow for searching and highlighting the overlapping polygon(s) [Kukulska et al., 2018]. Another tool named *check geometry validity* serves a similar purpose of linear and spatial objects (e.g. overlapping line segments) [Kukulska et al., 2018]. However it is not as suitable as *topology checker plug-in* regarding recognizing the overlaps of polygons.

As for the small gaps between two polygons, the *topology checker plug-in* tool does not always work well. It has an option named *must not have gaps*, which should principally guarantee that adjacent objects have no gaps and possess a common boundary [Kukulska et al., 2018]. Take the two polygons illustrated in the Figure 2.4 as an example, the *topology checker plug-in*

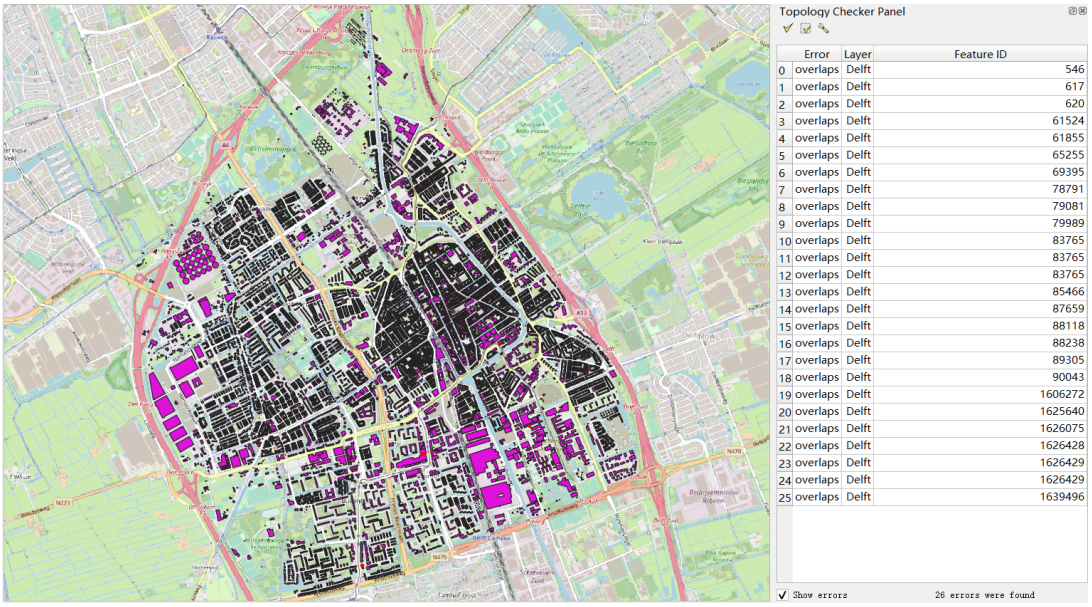


Figure 2.7: Use *topology checker plug-in* to recognize the overlapping polygon(s) in QGIS [QGIS Development Team, 2009], Delft region

is used to detect the small gap between these two polygons, yet no results can be found, as is shown in Figure 2.8.

2.4 Constrained Delaunay Triangulation

In this thesis, a constrained Delaunay triangulation is utilized as a supporting data structure to assist the snap rounding process. In order to get better acquainted with it, let us firstly start from the basic concept of a triangulation. A triangulation, is referring to a subdivision of a plane (currently in $\mathbb{R}^2$ space, can be expanded to $\mathbb{R}^3$ space) which contains a certain group of points and segments. If a set of points are given, the triangulation will be constructed by connecting them with straight line segments. These line segments intersect only at the corresponding endpoints [Lloyd, 1977]. The triangulation created via this approach is convex in practice [Ohori et al., 2012]. It is worth mentioning that in the triangulation there is one more face outside the convex hull (defined as the minimal convex shape that encloses all points), which is unbounded and is not of triangulated shape. The face is known to be *infinite face* of the triangulation.

Polygons can also be used as the basis to build the triangulation (actually this is used in the implementation of this thesis). The interior of a polygon is triangulated by creating line segments to connect vertices of the polygon but without passing through the exterior (or the boundary) of the polygon [o'Rourke, 1998]. That being said, the boundaries of the polygon becomes edges in the resulting triangulation and the interior of the polygon is subdivided into triangles. It is worth noting that in common practice, the convex hull of a polygon will be triangulated (including the interior of the polygon). For instance, in Figure 2.9 the

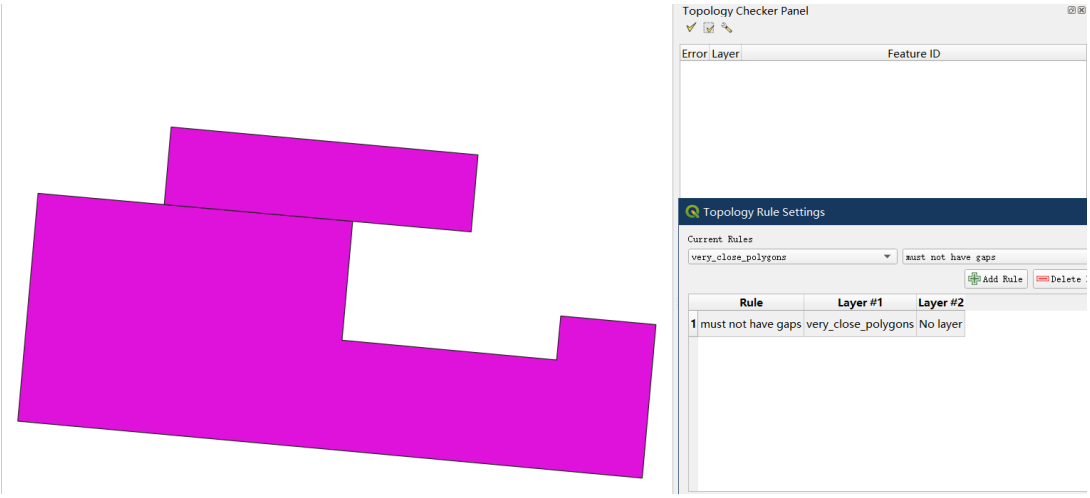
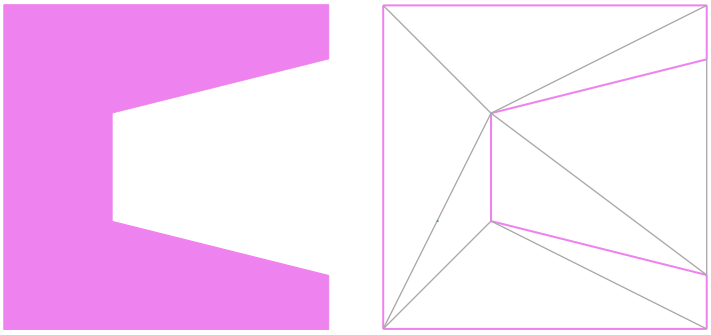


Figure 2.8: Use *topology checker plug-in* to recognize the small gap(s) between two close polygon(s) in QGIS [QGIS Development Team, 2009], the small gap shown in Figure 2.4b is not identified.



(a) A concave polygon. (b) The resulting triangulation.

Figure 2.9: Triangulation of a concave polygon.

input is a concave polygon but it is triangulated into a convex shape (see Figure 2.9b, the boundaries of the input polygon are highlighted in violet).

For an arbitrary polygon, it is always possible to construct a constrained triangulation for it. Note however, there can be several ways to triangulate a polygon. As is shown in Figure 2.10, different triangulations can be created based on the same quadrilateral. In Figure 2.10a, two skinny triangles are created (skinny means they are relatively "thin") while in Figure 2.10b the generated triangles are better-shaped. For many algorithms and applications, skinny structures can be potential problems thus well-shaped triangulations are preferred [Ohori et al., 2012]. A *Delaunay triangulation*, is a special kind of triangulation which is able to maximizes the minimum angles in the constructed triangulation. It is important to know that in a *Delaunay triangulation*, any three vertices that form a triangle have a circumcircle that contains no other points in it. This property is known as the *empty circle* property, which minimizes the likelihood of creating skewed shapes (i.e. elongated triangles).

(a) (b)

Figure 2.10: A quadrilateral is triangulated in two different ways. (a) A quadrilateral is triangulated into two skinny triangles. (b) A quadrilateral is triangulated into two better-shaped triangles comparing to (a).

One of the biggest characteristics of the *Delaunay triangulation* is that it would automatically adjust itself to satisfy certain properties in order to maintain its shape. However, in the context of this thesis, merely having the Delaunay property is not enough. Specifically, when creating a triangulation based on the input polygons, the boundaries of the polygons cannot be automatically changed (otherwise the shape and information of polygon would be lost), even if some skinny triangles may appear due to the presence of these boundaries. Therefore, a triangulation that allows fixed edges is essential in this thesis. A CT is such a triangulation that contains certain edges that can not be automatically changed. These edges are known to be *constrained edges*. They can not be changed in the triangulation unless users tells the triangulation to do so (i.e. inserting new constrained edges or removing existing constrained edges). With the consideration of combining the advantages of *Delaunay triangulation* and *constrained triangulation*, it is feasible to construct a *constrained Delaunay triangulation* (CDT) [Chew, 1987]. The CDT typically has the following properties:

- *Constrained edges* are ensured not to change automatically, such as unintended deletion.
- Changes are made locally (e.g. add or remove a *constrained edge*) therefore changing triangulation is a fast operation.
- The overall shape of a CDT is as close as possible to an ordinary *Delaunay triangulation*.

In the practice, a CDT usually provides good functionality to maintain a well-organized triangulated shape of the created triangulation, with robustness and efficiency. Having a CDT as a supporting data structure would help to easily handle the input polygons. A mature implementation is provided by the CGAL triangulation package [Boissonnat et al., 2000], which would later be used in the prototype of this thesis.

2.5 Geometry Repairing Based on a Triangulation

By utilizing a CT as a supporting data structure, [Ledoux et al., 2012] introduced a novel method to automatically fix geometries of possibly invalid polygons and to improve the robustness of valid ones. One of the biggest advantages of using a CT is that it allows for embedding both the geometry and the topology of the polygons at the same time, hence fewer operations need to be carried out when performing repairing process. The proposed method was developed based on the well-known CGAL library [Fabri and Pion, 2009], which

facilitates exact and inexact computations to enable robust and efficient arithmetic. It should be pointed out that the core part of the repairing method sorely includes a highly-efficient process which is known as *labelling triangles* and it does not demand any geometric calculations, thus it is fully robust and this has already been proven through extensive testing. The prototype (now it is mature enough as a great open-source software, called *prepair*) is available at https://github.com/tudelft3d/prepair.

Expanding upon the polygon repair, [Ohori et al., 2012] have proposed a new method to repair not only individual polygons but also planar partitions. In the developed approach, a CDT is utilized to help with the repairing process. Firstly all input polygons are validated and repaired (for invalid ones) using the schema mentioned in [Ledoux et al., 2012]. Subsequently the polygons are gradually added into a CDT, the triangles are then attached with tags indicating which polygons they should belong to (note that for the special area, e.g. the regions outside the domain, a stand-alone tag is used). With the tags as a basis, issues amongst polygons (e.g. two independently valid polygons but with an overlapping area) can be repaired by assigning the tags in accordance with certain criteria. At the end of this stage it is necessary to ensure that each triangle in the triangulation has only one tag assigned. After all repairing processes are completed, the final polygons are reconstructed from the triangulation and a valid planar partition is obtained.

2.6 Snap Rounding

In previous sections, the limitations and consequences of floating-point arithmetic have been described and discussed. What if users do want to avoid the potential issues of it as much as possible and at the same time ensure the reliability of the geometric computation results? There are currently two major solutions in practice: use exact arithmetic or use appropriate methods to generate geometric objects with finite-precision estimations. Exact arithmetic is known to be time-intensive and not suitable for large computations (as mentioned in the **Introduction**). The interest of this thesis is focused on the other substitution: finite-precision (or fixed-precision) approximation. SR is a representative of this method and it converts an arrangement of line segments from an arbitrary-precision representation into a lower fixed-precision form. It should however be noted that the original topology and geometry of the data would be slightly modified, this is known as a perturbation in SR [Raab, 1999].

Theoretically speaking, SR operation rounds the real number to the closest representable value with the given precision. Internally this is usually achieved by utilizing a grid (will be explained in details later). In other words, the snap rounded result has the same precision as the fixed-precision format, other extra precision in the original real number is thus lost. SR should be used with care in the practice as it will slightly (or radically, depending on the rounding *tolerance* used) modify the geometric and topological characteristics of the input data.

Figure 2.11 shows an example of SR [Halperin and Packer, 2002]. Figure 2.11a illustrates an arrangement consisting of arbitrary-precision represented line segments. SR typically proceeds by firstly creating a grid which contains all line segments. Each cell in the grid is called a *pixel*, the length of the cell is known to be the *resolution* of the grid, which actually indicates the *tolerance* of the SR operation. For instance, if the length of a cell is $1mm$, this implies the grid *resolution* and the *tolerance* are both equal to $1mm$. In the grid, if a cell contains a vertex then it is considered as a *hot pixel*. SR replaces all vertices by the center

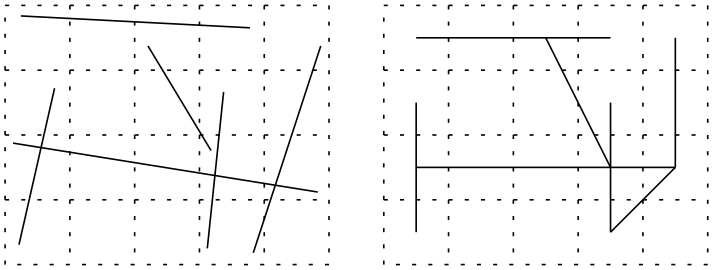


Figure 2.11: An arrangements of line segments before (a) and after (b) snap rounding [Halperin and Packer, 2002]

points of their corresponding *hot pixels*, as is shown in Figure 2.11b. Subsequently, all line segments are reorganized by connecting these center points. The intersections thus only happen in the centres of the grid cells rather than random places. SR operation normally produces better separated vertices and cleaner geometric arrangements compared to the initial input. It is worth noting that the rounded counterpart and the original arrangement shares similar topological properties, which suggests that SR does not change the topology to a large extent (if a proper *tolerance* is carefully chosen). Furthermore, SR can not only be applied in 2-dimensional space, but also be expanded to 3-dimensional space, see [Fortune, 1998] for more details.

2.7 Overview of Existed Algorithms Pertaining to Snap Rounding

The rounding scheme for a set of line segments was firstly introduced in [Greene and Yao, 1986]. The authors proposed a method to round a line segment arrangement, which creates a planar subdivision by dividing a two-dimensional space into smaller regions, to an integer grid. The proposed approach alters each line segment by moving each of its intersection point to the nearest point on the grid. Put it simply, each line segment is curved to a certain extent so that its intersections can align with the grid points and each line segment can be transformed to a polygonal path. Additionally the resulting polygonal path must be limited so that it would not pass over any other grid points. This constraint ensures that the polygonal path does not intersect or overlap with other grid points. Note that when each intersection point is moved to a grid point, it acts like a "hook" as it pulls other line segments that are connected to it to the same grid point. Such manner would keep the connected line segments well aligned with the grid and preserve the topological relationships. This is important since it allows the transformed polygonal path to retain its original shape and characteristics as much as possible while being more computationally efficient to store and manipulate.

The concept of SR was firstly given by Greene (in his unpublished manuscript *Integer line segment intersection* [Halperin and Packer, 2002]) and Hobby (in the published paper [Hobby, 1999]). The proposed algorithms make it simpler to perform rounding operations mentioned in [Greene and Yao, 1986] by disentangling the process of computing arrangement intersection vertices from the rounding process of line segments [Hobby, 1999]. See Figure 2.12 for

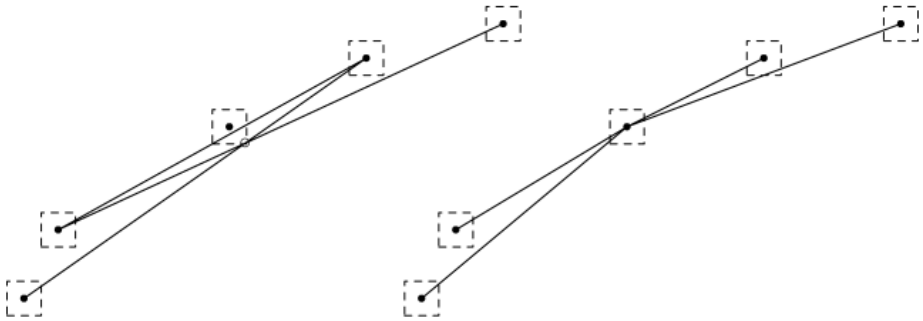


Figure 2.12: A snap rounding example given by [Hobby, 1999]. Left: Sample input with points rounded to the nearest grid points (marked by solid dots) and the tolerance squares (outlined by dashed lines). Right: The result of snapping segment(s) to the grid points.

an illustration. Firstly the Bentley-Ottmann sweep line algorithm [Bentley and Ottmann, 1979] is utilized to locate the intersection points in the input. Afterwards the endpoints and intersection points of all line segments would be rounded to the corresponding nearest grid points, as is shown in the left figure of Figure 2.12. The solid dots stand for the rounded points (nearest grid points, not original input points), the dashed lines around them represent the tolerance squares. If a segment s intersects a tolerance square of a grid point p, it will be altered by being slightly bent so as to pass through p. This is known to be *snapping segment to point*, as is shown in the right figure of Figure 2.12.

The algorithm presented in [Hobby, 1999] has its foundation in finite-precision arithmetic and has been proven to be robust. However, the limitation lies in its lack of being dynamic, more specifically, all the input segments are assumed to be given at once. Such scheme does not allow for incremental updates of the line segments, that being said, a global reprocessing may need to be performed each time a small change is made. In practice there can be some applications where the input line segments may change frequently, which will possibly cause the computational cost too prohibitive for users to accept. In order to overcome this issue, a novel method was proposed in [Guibas and Marimont, 1995] to robustly and dynamically perform the snap rounding of an arrangement of line segments. Their method includes an additional algorithm described in [Mulmuley, 1994] which provides a data structure, the vertical cell decomposition (VCD) of a set of line segments, to allow efficiently and dynamically locating, inserting and deleting the line segments. Guibas and Marimont combine the robustness of the algorithm provided by Hobby and the capability of dynamic processing of the Mulmuley's algorithm. Besides, they also provided basic proofs of the certain topological characteristics and properties maintained by SR.

Based on the previous work, Goodrich et al. provided improved SR algorithms with higher efficiency. They gave two algorithms which are output-sensitive for rounding a set of line segments in $\mathbb{R}^2$ with a high degree of efficiency. Additionally a basic framework for performing snap rounding in 3-dimensional space ($\mathbb{R}^3$) was also included. As for SR in 2-dimensional space, a determinstic (the output is the same if given the same input) algorithm was presented based on the plane-sweep method mentioned in [Bentley and Ottmann, 1979] and a randomized (the output can be different for the same input) method was described on the basis of randomized incremental construction (known as RIC). Note that in the randomized algorithm an ordered list (e.g. a binary search tree) is maintained to help to keep track of

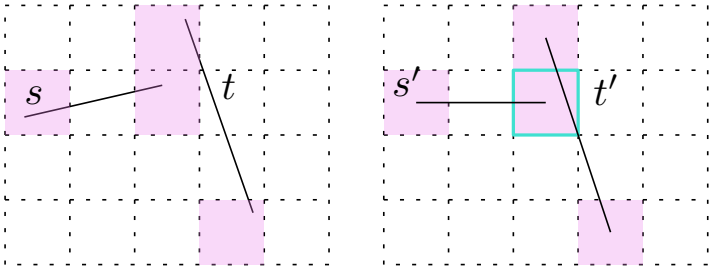


Figure 2.13: A vertex becomes very close to a non-incident edge after (b) snap rounding. The figure is derived from [Halperin and Packer, 2002]

intersections, which significantly improves the efficiency of SR especially when the number of line segments intersecting in a pixel is relatively large.

Iterated Snap Rounding (ISR for shorthand) is a variation of SR, which was firstly introduced in [Halperin and Packer, 2002]. It was proposed to solve the issue of possible near-degenerate output which could be produced by standard SR. As previously illustrated in Figure 2.11, standard SR rounds the vertices by replacing them using the central point of *hot pixels*. However, in the case of large input coordinates, the distance between a rounded vertex and a rounded non-incident edge can be very close compared to the length of the cell in the grid, this is considered a near-degenerate output. See Figure 2.13 for an straightforward illustration. Two line segments s and t are displayed in Figure 2.13a, the *hot pixels* (recall that in the grid, *hot pixels* refer to the pixels that contains the endpoints or intersection points of input line segments) are highlighted in violet. The rounded counterparts are denoted as s' and t' respectively, as is shown in Figure 2.13b. Observing from the figure, t' passes through a *hot pixel* (the boundary is highlighted in turquoise), which indicates that the endpoint of s' on the right side is very close to t'.

ISR solves the possible near-degenerate output issue mentioned above by utilizing SR multiple times (repeatedly) until each rounded vertex is at least half-a-unit (typically the length of one side of a cell in the grid) away from any rounded non-incident edge. Put it differently, ISR ensures that in the rounded arrangement, there are no rounded line segments intersecting any *hot pixels*. Comparing to the standard SR, ISR enhances the robustness of the rounded counterpart of the input line segment arrangement for data manipulation using finite-precision arithmetic hence improves the stability of SR. However, ISR requires multiple executions of standard SR operations, hence it can be slower than the ordinary SR. What's more, the result produced by ISR may not be accurate enough, specifically, the rounded counterpart of the input can be shifted or distorted away from its original position due to the iterative process. An illustrative example is shown in Figure 2.14.

For the purpose of solving the inaccuracy issue of ISR, *Iterated Snap Rounding with Bounded Drift* (ISRBD) was proposed in [Packer, 2006]. The authors aimed to solve the limitations of standard SR (possible degenerate output) and ISR (rounded result can drift to a large extent compared to the input). ISRBD ensures that the deviation between the input line segment arrangement and its rounded counterpart is less than a predefined value. The core idea of ISRBD is to introduce a small amount of new *hot pixels* to limit the magnitude of rounding. Such manner would help to constrain the shift of the original input line segments to a predefined amount. Albeit the construction and utilization of the new *hot pixels* would

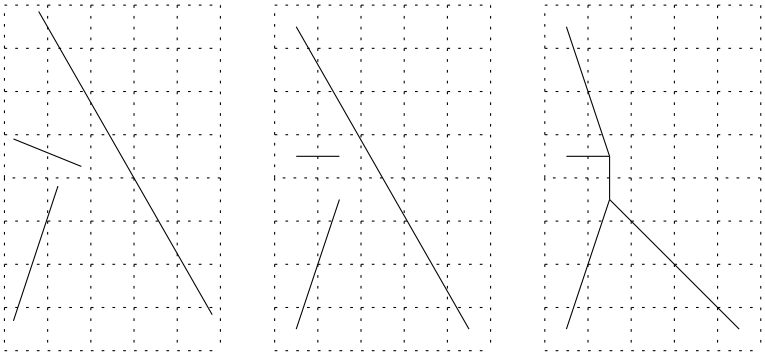


Figure 2.14: (a) An arrangement of segments before. (b) After Snap Rounding. (c) After Iterated Snap Rounding [Halperin and Packer, 2002].

not require a large amount of additional time (comparing to ISR), the overall run time is still long. Another limitation is that the output of ISRBD can be noncanonical, this is because the locations of newly-added *hot pixels* are decided by the proposed algorithm so as to satisfy the aforementioned rounding magnitude.

Bearing these in mind, a new scheme named *stable snap rounding* was introduced in [Hershberger, 2011], it was based on the conception of SR but actually alters the definition of SR that has previously been discussed (using a grid and the *hot pixels* to round the line segments). The authors proposed a new approach to utilize the *hot pixels*, they basically split the *hot pixels* into two different kinds of categories and applied disparate processing strategies in order to take control of the deformation of the input line segments. The biggest advantage comparing to ISRBD or ISR is that the *stable snap rounding* is idempotent. Particularly, if SR operation is performed repeatedly on the same dataset, the result will not change. *Stable snap rounding* improves the robustness and stability of ISRBD and ISR, note however, it does not guarantee to eliminate near-degenerate cases while ISRBD (and ISR) does.

Although the aforementioned methods have shown promising results, there is still room for improvement. Building upon the concepts of SR, recent studies in the field have proposed a new approach which is called *Snap Rounding with Restore* (SRR) [Belussi et al., 2016]. Comparing to ISRBD and ISR, the most noticeable modification is that for the situation where a vertex is too near to a non-incident edge after rounding, the non-incident edge is moved in the opposite direction (this means it is being moved back towards its original location) instead of performing a SR operation (snapping the vertex and the non-incident edge together). This process is known as *restore* in the proposed method, which possesses the capability to improve the quality of the output whereas it may increase the time complexity. SRR can improve the robustness of the result and at the same time maintain the topological characteristics as well as possible, therefore SRR is well suitable for the application scenarios where both robustness and preservation of topological properties are considered important.

2.8 Summarize

Up to this point, the issues of applying floating-point arithmetic regarding the geometric algorithm implementations and other possible errors in the real-world datasets have been discussed. Furthermore, the basic concepts of different triangulations and snap rounding (SR) are introduced as well. It should be pointed out that the precision issues of floating-point arithmetic and the possibly degenerate input data are correlated to a certain degree. That is to say, input datasets would cause faulty output (e.g. containing topological errors) in combination of using floating-point numbers [Raab, 1999].

Based on the content above, following conclusions can be derived:

- Using floating-point arithmetic can raise the round off errors of coordinates, which will lead to unintended shape changes of geometric objects (e.g. polygons) and possibly topological errors.
- Datasets commonly contain topological errors such as small gaps and overlapping area between polygons not only because of the round off errors but also the limitations of data acquisition, storage, management and exchange (e.g. converting datasets from one file format to another file format).
- Round off errors and topological errors are correlated, they would both affect the robustness and reliability of relevant geometric algorithms and downstream analyses.
- Snap rounding (SR) is a commonly-used technique to convert arbitrary-precision representations into finite-precision representations in order to avoid the issues caused by round off errors. Existing SR implementations (e.g. the iterated snap rounding described in [Halperin and Packer, 2002] and implemented in CGAL [Fabri and Pion, 2009]) mainly focus on the line segments. Note that this approach can be modified and extended to be applicable to polygons as well (using the boundaries of polygons as input line segments).
- Constrained triangulation (CT) can be used as an auxiliary data structure to robustly deal with geometric objects especially for polygons. It also enables fully-automatic repairing of certain topological errors (see [Ledoux et al., 2012] and [Ohori et al., 2012]).

Having these in mind, in this thesis, an robust approach is intended to be developed with the purpose of performing SR of polygons in 2-dimensional space, using CT as a supporting data structure. The main interest lies in the topological relations of polygons and focuses on the cases that would possibly cause the invalidities (e.g. topological errors, algorithm crash, infinite loop, etc.). Specifically, dealing with the following two different cases is the main objective of the thesis: (i) close vertices, i.e. the distance between two vertices is below a certain threshold; (ii) close vertices and boundaries, i.e. the distance between a vertex and a boundary (belonging to a polygon) is below a certain threshold. The input and output would mainly be polygons, with the potential to expand into the combination of polygons, line segments and points.

3 Methodology

3.1 Overview

In this chapter the triangulation-based methodology will be proposed and elaborated with details in accordance with aforementioned research objectives. The method overview is depicted in Figure 3.1.

Firstly, an input file is opened and read. The file is typically in GeoPackage (GPKG) format and contains a certain number of polygons. When loading each polygon into the memory, one can choose to shift the coordinates of vertices by subtracting the minimum x and y values of the dataset. This would help to reduce the magnitude of the coordinates and at the same time preserve the precision as much as possible.

Secondly, the CGAL triangulation package [Boissonnat et al., 2000] is used to build a CDT, with the exterior and interior boundaries of polygons being the constrained edges. CDT is known to be able to make the triangulation as well shaped as possible and would improve the efficiency when finding the vertex to boundary case (see Figure 3.18), as it by definition can avoid the skinny triangles wherever possible.

Subsequently, the whole triangulation is tagged by adding tags to the triangles (faces) and constructing constraints accordingly. For each constrained edge, a constraint is built with the ID information indicating which polygon(s) it should belong to and stored in an individual container. The stored constraints represent the boundaries of polygons, which will be later used in the SR process. At the end of this stage, each constraint will have at least one tag. Constraints with more than one tags indicate that they are the common boundaries between polygons or they are part of the overlapping area.

After having tagged the triangulation, SR is performed on the input polygons. Given a predefined tolerance (e.g. $0.01m$), there can be multiple cases that need to be snap rounded in the dataset. SR starts from the case with minimum distance first, then gradually process the rest until no other cases can be found under this tolerance. In such manner, cascading effects can be avoided to a certain extent. For instance, if SR executes from a case with a relatively large distance first, the rounding operation would possibly create new cases that require to be processed again. However, if SR proceeds from the case with minimum distance, the changes to be made would have minimal impact on other parts of the triangulation and thus can avoid the cascading effects as much as possible.

Last but not least, polygons would be reconstructed from the container which holds all constraints. The ID information of constraints would be used to mark the ID of the reconstructed polygons. Finally the polygons would be output to a GPKG file.

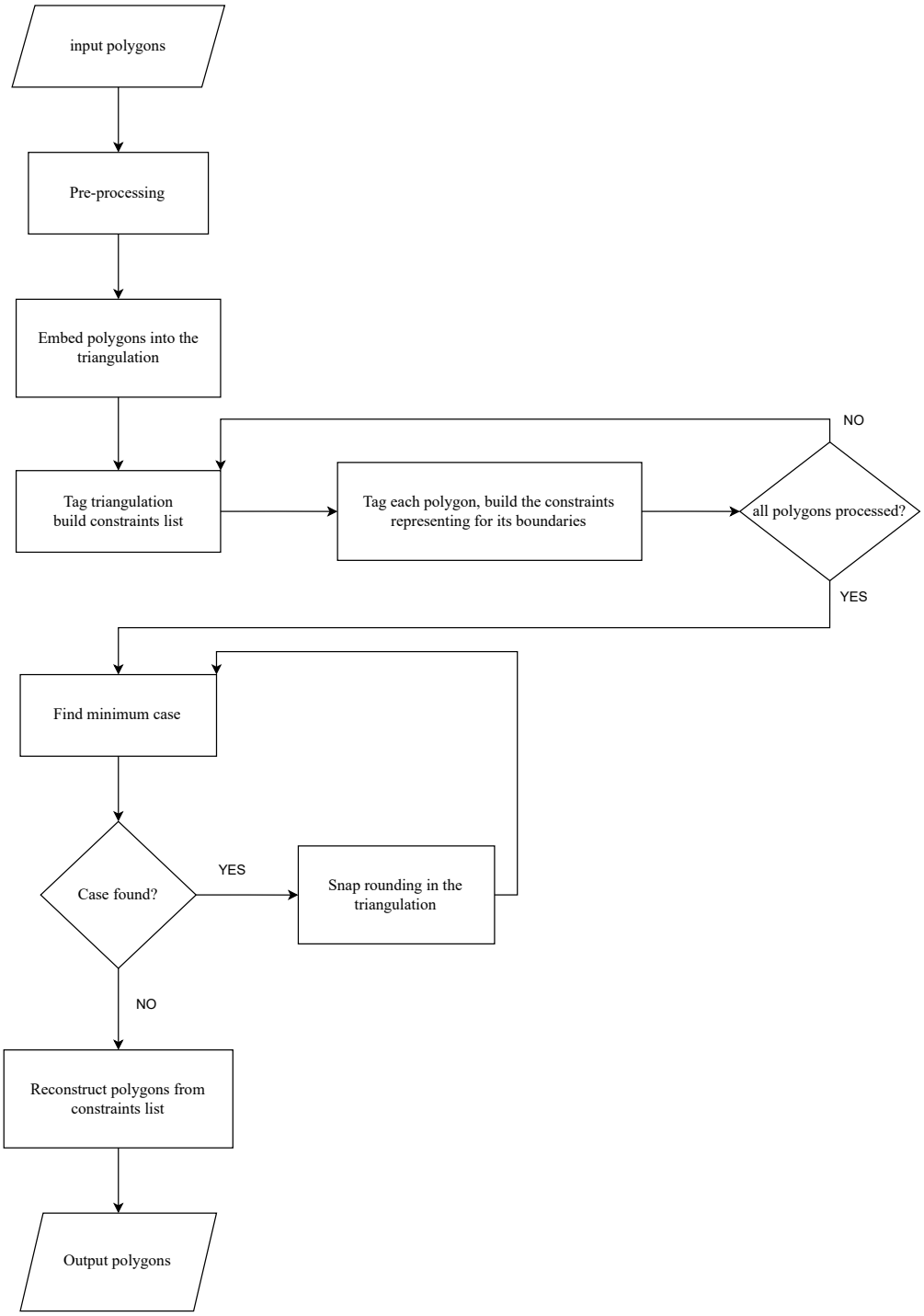


Figure 3.1: Methodology overview

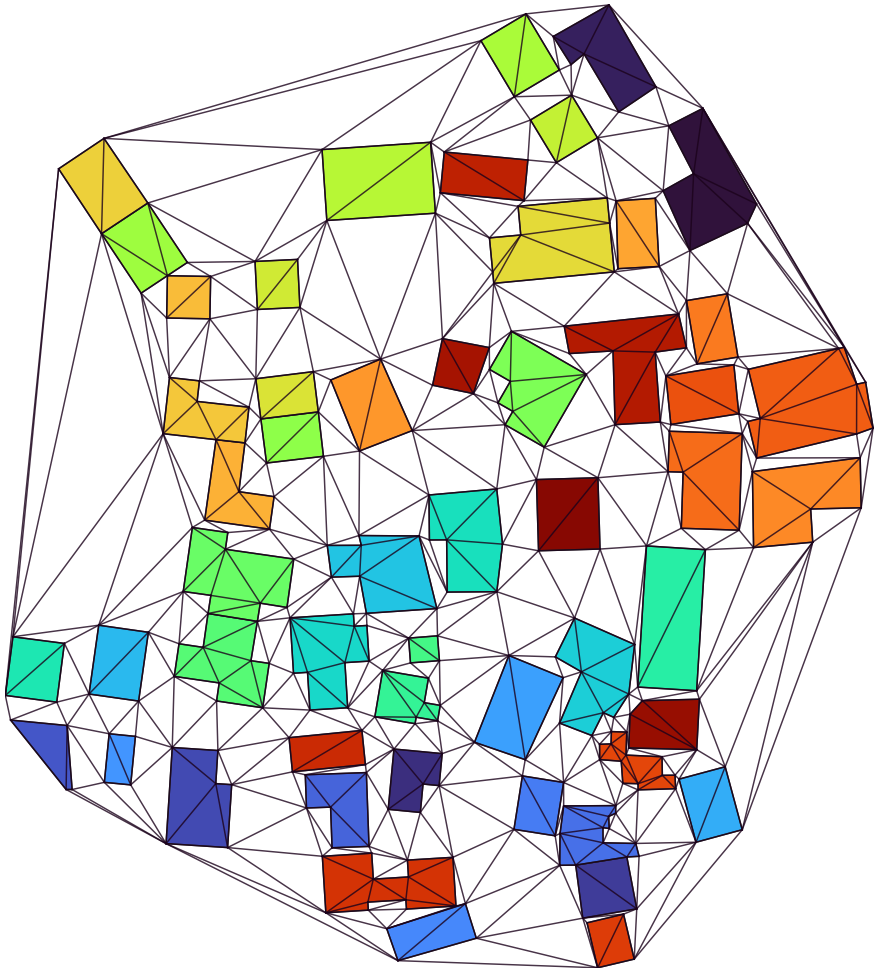


Figure 3.2: Embed polygons in a triangulation

3.2 Embed Polygons to a Constrained Delaunay Triangulation

In order to manage and manipulate the input geometries, a CDT is used as a basic data structure to hold all necessary elements (i.e. polygons, line segments and vertices). Constrained edges should be handled with care in order to keep track of the attributes while SR is being performed.

Figure 3.2 depicts an example. Polygons are embedded in a CDT and their boundaries are used as constraints. It is worth noting that, a CDT is constructed as close as possible to a Delaunay triangulation (DT) [Chew, 1987]. However, in some area, there can still be very skinny triangles due to the existence of constrained edges.

The embedding process is described in Algorithm 3.1.

Algorithm 3.1: EMBED POLYGONS($\mathcal{S}_p$, CDT)

Input: A set of polygons $\mathcal{S}_p$
Output: A constrained Delaunay triangulation CDT

```
1 for P in S_p do
2     for edge in exterior of P do
3         insert edge to CDT as a constraint;
      // if containing holes, insert as constraints
4     if P contains holes then
5         for hole in holes do
              // do not consider nested holes
6             for edge in exterior of hole do
7                 insert edge to CDT as a constraint;
8 return CDT
```

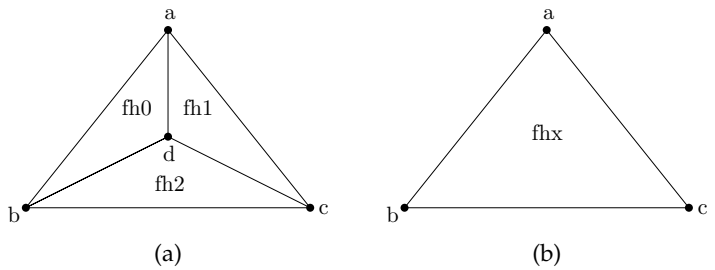


Figure 3.3: A simple triangulation. (a) Before removing vertex d. (b) After removing vertex d.

3.3 Tagging the Triangulation to Extract Polygon Boundaries

After the polygons have been embedded into a triangulation, it is of much importance to know which polygons the triangulation elements should belong to. In other words, one must be able to identify the polygons associated with their id information so as to reconstruct polygons from the triangulation in the subsequent steps. This is the basis where the idea of tagging lies on. The id information of input polygons are used to add tags to the triangulation. Tagged elements would further allow to keep track of the changes and update the polygon boundaries accordingly.

Triangulation is automatically managed by CGAL 2D Triangulations package [Boissonnat et al., 2000]. Necessary components can be accessed by `handles` (internally it can be considered as *pointers*), such as `Face_handle` for accessing triangles in the triangulation and `Vertex_handle` to obtain the vertices. However, `handles` are not always considered safe to use, especially when the triangulation is being dynamically altered. Insertion and removal of vertices would probably invalidate the original `handles` to a small / large extent. For instance, suppose a central point is going to be removed, after the removal the triangulation would be updated automatically and the ID information of original `Face_handles` will be lost. This makes it difficult to use `handles` to follow the ID changes. Take Figure 3.3 for an

example. Assume there are three polygons $\triangle abd$, $\triangle adc$ and $\triangle dbc$. The `Face_handles` are `fh0`, `fh1` and `fh2` respectively. The three triangles also have their own different ids which can be queried via three `Face_handles`, as is shown in the figure. Since three individual triangles are touching each other, all of the edges presented are constrained (representing boundaries). Now suppose vertex `d` is intended to be removed, the constrained edges `ad`, `bd`, `cd` would all be removed as well. One of the `Face_handles` might be preserved to represent the $\triangle abc$, yet the other two `Face_handles` would become invalid and the ID information would be lost.

With the aim of avoiding the problems mentioned above, a pragmatic solution has been implemented to keep track of the changes simultaneously by maintaining a separate data structure. Since `handles` are not stable when the triangulation is being modified, one can alternatively store and trace the elements which will not be mechanically changed by the triangulation package. The constrained edges, or constraints, are a great entity to realize this idea. Constraints in the triangulation stand for the boundaries (including exterior and interior), once they are tagged with the ID information, manual intervention is utilized to deliberately amend the stored constraints. Put it differently, one can decide how the constraints are revised in accordance with the changes made in the triangulation and be aware of the ongoing modifications.

The underlying method of the tagging process is Breadth-first search (BFS). This will be explained later in Section 3.3.1. The goal of this step would be to obtain a set of constraints with the id information from corresponding polygons attached.

3.3.1 Apply Breadth-first search for tagging

Breadth-first search, also known as BFS, is a commonly and widely used algorithm to search for a certain element in a data structure (e.g. a binary search tree) which satisfies specific conditions [Bundy and Wallen, 1984]. It can also be applied on simulating the expansion process within a predefined range, e.g. flood fill. In this sense Depth-first search (DFS) shares similar suitability [Tjiharjadi and Setiawan, 2016]. The aforementioned tagging process can be reasonably regarded as a variation of flood fill. It has been firstly used for tagging the triangulation in [Ohori et al., 2012]. Based on that, similar process is applied for tagging in this thesis but with some modifications. For an individual polygon, the process starts from a certain face, then the affiliated constraints are constructed with the ID information of the polygon and stored in a container. The boundaries of the polygons are deemed as the restricted area where the tagging process can not reach. It is important to know that the restrictions consist of not only the exterior boundaries but also the possible interior boundaries.

Both BFS and DFS can be utilized to implement the tagging procedure, however, BFS is preferred as the fundamental method although DFS can be more efficient in some cases. The main concern is DFS would require a stack to keep all necessary visiting elements. A stack overflow might arise if the DFS stack is too deep, e.g. when the triangulation is very large and complex. The prototype of this thesis is expected to possess the capability of working with at least middle-sized dataset, and preferably large dataset (e.g. millions of polygons). In light of this, robustness would come first over the consideration of efficiency.

Figure 3.5 illustrates the process of tagging. The overall shape of input polygon is shown in Figure 3.5a, which contains an interior hole (p_4, p_5, p_6, p_7). The triangulation with the

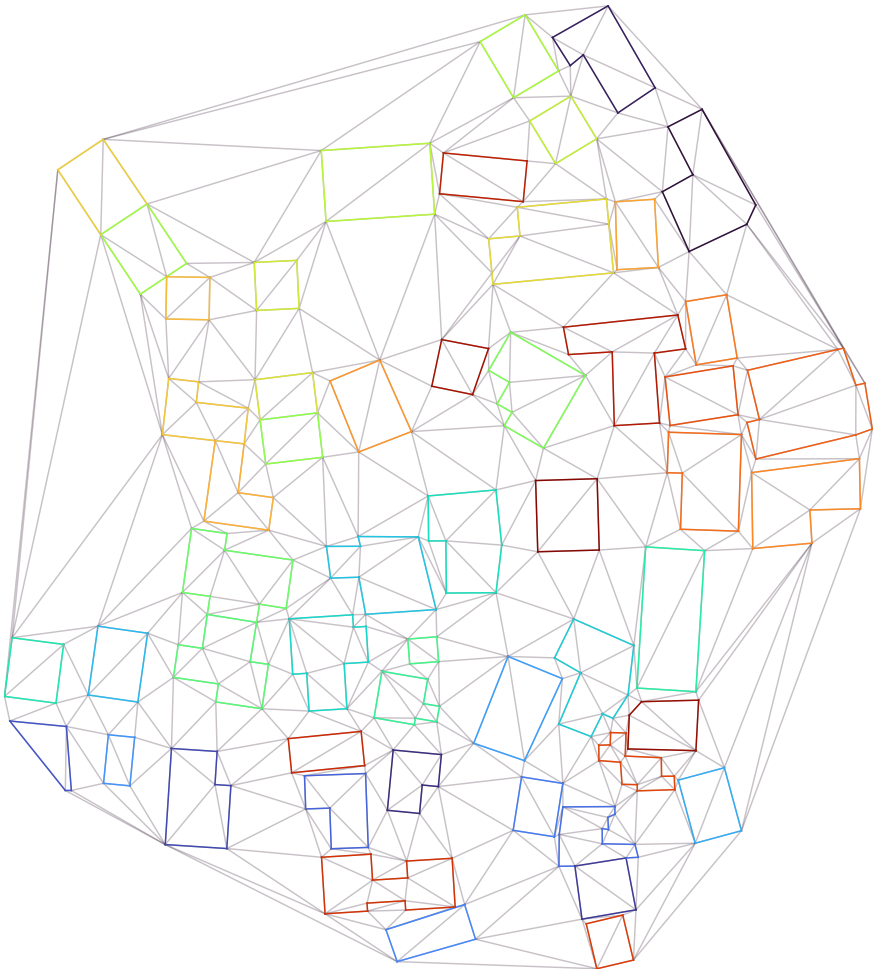
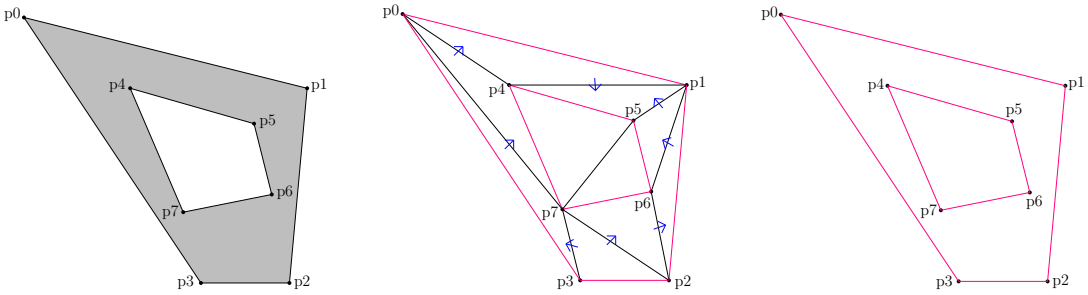
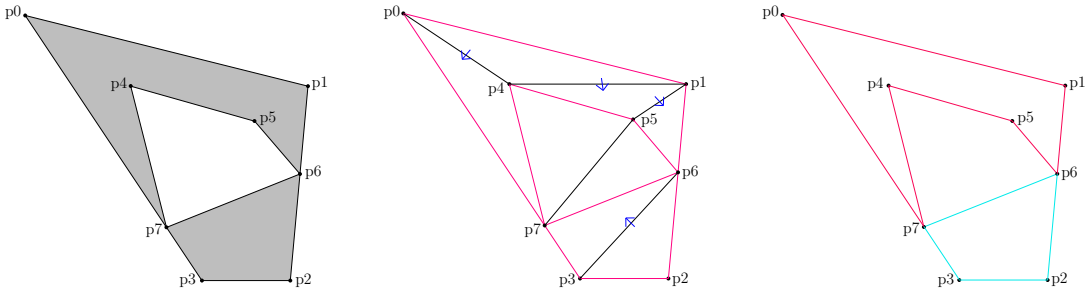


Figure 3.4: Tagged constraints



(a) input polygon (b) tagging process (c) tagging result

Figure 3.5: Tagging example.



(a) input polygon (b) tagging process (c) tagging result

Figure 3.6: Tagging example (with faulty input).

polygon embedded is depicted in Figure 3.5b, constraints are highlighted in pink. Suppose $\triangle p_7p_3p_2$ is selected as the starting face, from which the tagging process is expanded gradually until constraints are encountered. For each face which can be reachable from the starting face, all the three incident edges are inspected for being constrained or not. Constraints are constructed for the constrained edges, supplemented by the ID information of the polygon it belongs to. Figure 3.5c shows the result of the tagging stage, the boundaries, including exterior and interior(s) are obtained and stored in a container.

For real-world datasets, extra care needs to be taken since there are plenty invalid geometries in them. Although dealing with invalidity is not the main goal of this thesis and hence out of range, the capability of accepting possibly faulty input without program crashing will still be deemed as an asset for the prototype. To elucidate this, refer to Figure 3.6 for an example.

Figure 3.6a shows an invalid geometry, a polygon with its interior hole touching the exterior boundary. This irregular shape will result in the existence of unconnected parts regarding the polygon interior. In Figure 3.6b, suppose the tagging is initialised from the bottom face ($\triangle p_6p_3p_2$), the expansion will stop by the constraint p_7p_6. The upper part of the polygon is not accessible from the selected starting face, hence it will remain untagged. The result of tagging in this case is only the lower part of the input, which is highlighted in cyan in Figure 3.5c. Conversely, if the selected starting face is located on the top of the polygon (i.e. $\triangle p_0p_4p_1$), the tagging process will also stop at the constraint p_7p_6, leading to the result highlighted in magenta in Figure 3.6c.

For the purpose of handling the situations described above, using multiple starting faces is a feasible way and relatively straightforward to implement. Taking Figure 3.6b for an example, both $\triangle\ p_0p_4p_1$ and $\triangle\ p_6p_3p_2$ are used as possibly starting faces for initialising the tagging. All of the constrained edges representing exterior and interior(s) can be accessible. The tagging result would be the conjunction of magenta part and the cyan part in Figure 3.6c.

In the practice, there can be other complex arrangements that would cause problems such as overlap, self-intersection, etc. In order to make the algorithm as robust as possible, all incident faces of each vertex on the exterior boundary of a polygon would be firstly investigated, and the starting face (of this polygon) is selected based on the criteria described in Section 3.3.2.

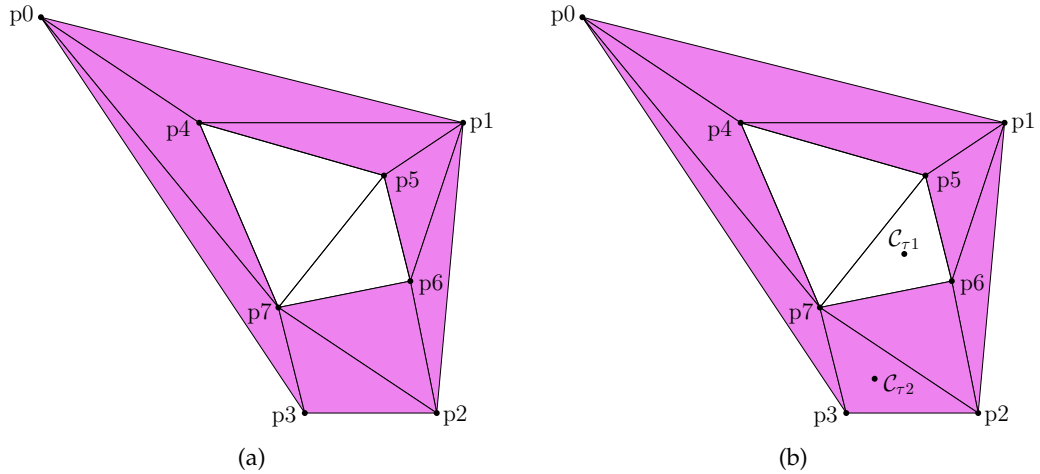


Figure 3.7: Exterior and interior triangles of a polygon. (a) A polygon with a hole. The interior triangles are highlighted in violet. (b) Two centroids, $\mathcal{C}_{\tau 1}$ and $\mathcal{C}_{\tau 2}$ are shown. $\mathcal{C}_{\tau 1}$ is outside the polygon, so does the triangle τ_1 ($\triangle\ p_5p_6p_7$). $\mathcal{C}_{\tau 2}$ is inside the polygon, the triangle τ_2 ($\triangle\ p_7p_2p_3$) is also inside the polygon.

3.3.2 Find starting face

Since BFS is being used for tagging, it is indispensable to define and locate the starting face, where the BFS process will begin. In Section 3.3.1 it has been discussed about how to apply BFS for tagging. How proper starting faces can be found in the triangulation will be discussed in this section.

In the triangulation, triangles are formed and connected by either unconstrained edges or constrained edges. For the polygons that have previously been embedded, the constrained edges representing polygonal boundaries will form multiple closed rings. The triangles inside these rings indicate that they are related to the polygon which contains them, hence the ID information of this polygon can be assigned to these triangles based on their actual locations. It should be pointed out that, considering the possibility that a polygon can have one or multiple hole(s), care needs to be taken when determining if a triangle is inside a polygon or not. See Figure 3.7a for an example, triangle τ_1 ($\triangle\ p_5p_6p_7$) is not inside the polygon since it is actually in the hole hence not part of the interior. Triangle τ_2 ($\triangle\ p_7p_2p_3$) is considered inside the polygon since it is part of the interior. Having these in mind, the following conditions (to determine whether a triangle is inside a polygon) are proposed:

- **Condition 1**: The triangle must be inside the exterior boundary of the polygon.
- **Condition 2**: The triangle must be outside any possible interior holes of the polygon.

Only when these two conditions are met at the same time will a triangle be deemed inside a polygon. Note that nested holes (i.e. hole containing hole(s)) are not taken into consideration as it is less common in the datasets.

Based on the conditions summarized above, the next objective is to simplify them, making them more straightforward for easier and clearer implementation later on. One pragmatic way is to use the centroid of a triangle to determine its location. The centroid is calculated

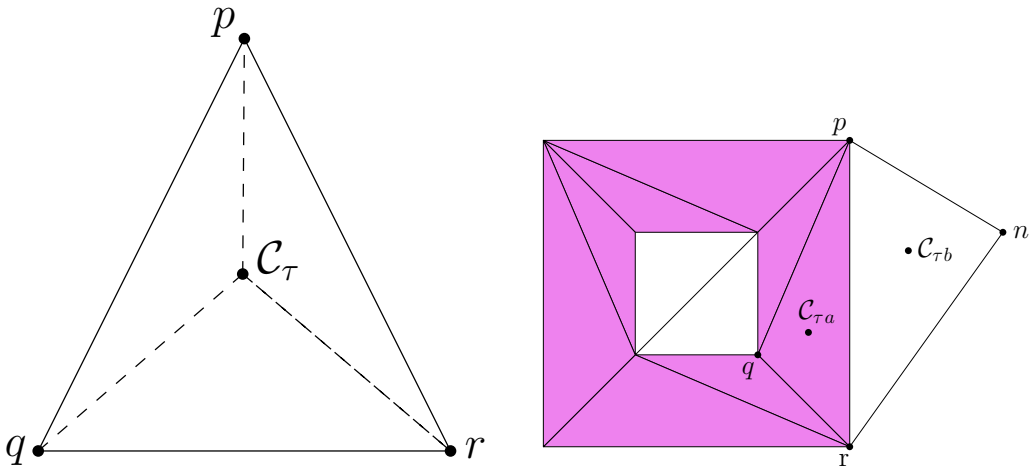


Figure 3.8: Centroid of a triangle. (a) Centroid of the triangle τ ($\triangle$ pqr). (b) A polygon with its interior highlighted in violet. Triangle τ_a ($\triangle$ pqr) is inside the polygon and its centroid $\mathcal{C}_{\tau a}$ is also inside the polygon. Triangle τ_b ($\triangle$ prn) is outside the polygon with its centroid $\mathcal{C}_{\tau b}$ being outside the polygon.

in accordance with the coordinates of the three vertices of the triangle. To be more specific, the centroid represents the average position of all three vertices in a triangle. Suppose there exists a triangle τ ($\triangle$ pqr) as is shown in Figure 3.8a, its centroid can be computed using the following formula:

$$\mathcal{C}_\tau = \left(\frac{x_p + x_q + x_r}{3}, \frac{y_p + y_q + y_r}{3} \right). \tag{3.1}$$

Bearing these in mind, a conclusion can be drawn that if a triangle is inside a polygon, then its centroid must be inside the polygon as well, and *vice versa*. See Figure 3.8b for an illustration. A polygon contains a hole in the middle and all triangles in the interior are highlighted in violet. Let us focus on the two rightmost triangles: τ_a ($\triangle$ pqr) and τ_b ($\triangle$ prn), with $\mathcal{C}_{\tau a}$ and $\mathcal{C}_{\tau b}$ being the centroids respectively. $\mathcal{C}_{\tau a}$ is located inside the polygon and the triangle containing it is also inside the polygon. On the contrary, $\mathcal{C}_{\tau b}$ is located outside the polygon and so does the triangle containing it. By querying the location of a centroid, one can easily know the location of a triangle. Note that the location here means either inside or outside a polygon. It should be emphasized that the centroid of a triangle should not be exactly on the boundary, as it indicates the triangle containing it is collapsed to a line segment (e.g. if $\mathcal{C}_{\tau a}$ is exactly located on the boundary pr, then it means triangle τ_a is collapsed to the line segment pr).

After clarifying that the centroid can be used to determine the location of a triangle (relative to a polygon), now the conditions described above can be refined as:

- **Condition 1**: The centroid of a triangle must be inside the exterior of the polygon.
- **Condition 2**: The centroid of a triangle must be outside any possible hole(s).

As is shown in Figure 3.7b, the centroid $\mathcal{C}_{\tau 2}$ of triangle τ_2 is inside the exterior ring $\{p_0, p_1, p_2, p_3, p_0\}$. Meanwhile, the centroid $\mathcal{C}_{\tau 2}$ is outside the interior hole $\{p_4, p_5, p_6, p_7, p_4\}$. The two conditions are satisfied and thus $\mathcal{C}_{\tau 2}$ is considered inside the polygon, which tells us the triangle containing it (τ_2) is also inside the polygon. In contrast, the centroid $\mathcal{C}_{\tau 1}$ is inside not only the exterior ring but also the interior hole. This indicates that $\mathcal{C}_{\tau 1}$ is actually outside the polygon, and the triangle it belongs to (τ_1) is in the exterior of the polygon.

Up to this point, the location of a triangle (in the triangulation) relative to a polygon has been successfully queried. In the triangulation, a starting face (i.e. a triangle in the interior of a polygon) can then be obtained by firstly querying the incident faces (triangles) of a vertex and then filtering out the triangles which satisfy the two conditions summarized above. Furthermore, it is also necessary to traverse each vertex in the exterior ring of a polygon when querying the incident faces so as to achieve a relatively good level of robustness (see Section 3.3.1 and Figure 3.6).

Based on this section and previous sections in this chapter, the overview of the tagging triangulation is provided in Algorithm 3.2.

Algorithm 3.2: TAG TRIANGULATION ($\mathcal{S}_p$, CDT, $\mathcal{L}_c$, CDT_t)

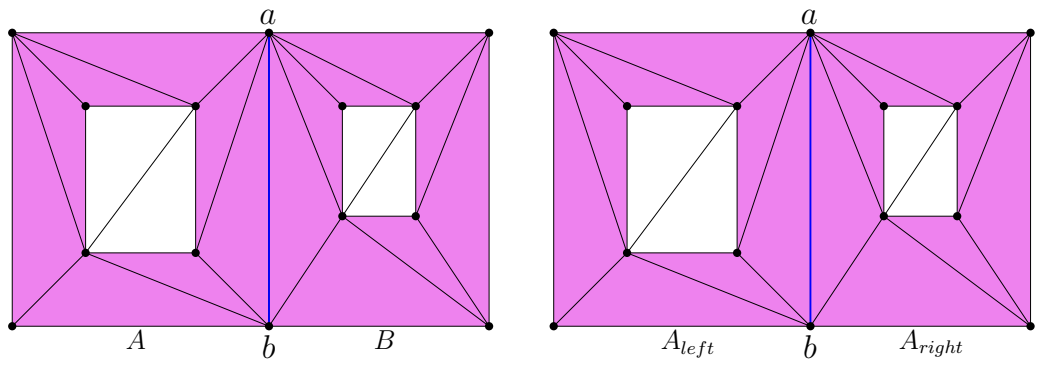
Input: A set of polygons $\mathcal{S}_p$, A constrained Delaunay triangulation CDT
Output: A list $\mathcal{L}_c$ to store all constraints (boundaries of polygons), A tagged constrained Delaunay triangulation CDT_t

```
1  for P in S_p do
2      V ← to store all starting faces of one polygon;
3      for point in exterior of P do
4          τ ← get each incident face of point;
5          σ ← get the centroid of τ;
6          if σ inside P then
7              add τ to V;
8      if V not empty then
9          for τ in V do
               // perform BFS tagging on one polygon
               // after this operation L_c will be populated
               // CDT will be tagged and become CDT_t
10             perform BFS tagging on P using τ;
11 return L_c, CDT_t
```

3.3.3 Merging process for common boundaries

During the tagging process, constraints representing boundaries are gradually built and stored in a container. For the common boundaries, the ID information is merged in order to allow a boundary to have multiple ID information. As is shown in Figure 3.9a.

Polygon A and B are touching with the line segment ab being the common boundary. Suppose the tagging process is initialized from polygon A first, the constraint ab will be built and stored, the ID information will be assigned the value A. Afterwards, when polygon B



(a) (b)

Figure 3.9: Common boundary between two polygons. (a) A and B are adjacent polygons with common boundary ab. (b) A_{left} and A_{right} have the same polygon ID (A) in the dataset, but they are disconnected and should be considered as two polygons.

is being tagged, ab will also be traversed and will be assigned the value B. However, the constraint ab has already existed in the container. It is preferred not to store repeated constraints in the container as it may cause problems in the reconstruction stage. Furthermore, repeated storage would increase the required space resources as well. In order to avoid this, the ID information of both polygons is merged for one constraint instead of constructing two separate constraints with different ID information. In Figure 3.9a, the constraint ab will firstly be assigned ID A and then its ID field will be merged with ID B, resulting in ID field containing A and B at the same time. It is worth noting that in practice, there can exist various invalid polygons such as disconnected parts (depicted in Figure 3.9b). In such case the constraint ab representing the common boundary between the left and right parts of A (denoted by A_{left} and A_{right}) will only contain one ID (A), rather than storing A twice.

The merging process can be robustly implemented by utilizing an unordered set (or a dictionary alternatively), the details are provided in Algorithm 3.3.

Algorithm 3.3: MERGE ID INFORMATION ($\mathcal{V}_{lhs}$, $\mathcal{V}_{rhs}$)

Input: A vector $\mathcal{V}_{lhs}$ to hold the current ID information, A vector $\mathcal{V}_{rhs}$ to hold the ID information to be merged

Output: The vector $\mathcal{V}_{lhs}$ which merges the ID information from $\mathcal{V}_{rhs}$ via a non-duplicate manner

```
S_lhs ← to store all ID information of V_lhs;
for id in V_lhs do
    insert id to S_lhs;
for id_rhs in V_rhs do
    if id_rhs not in S_lhs then
        add id_rhs to V_lhs;
return V_lhs
```

Now all aspects of the tagging stage have been discussed. To summarize, firstly the input polygons are embedded into the triangulation with their boundaries as constrained edges.

Afterwards, for each polygon, each vertex in its exterior ring is traversed to obtain the incident triangles (faces). The centroids of these triangles are then computed and for those that are inside the polygon, the corresponding triangles are stored as potentially starting faces. Next for each polygon, the stored starting faces are utilized to perform a BFS expansion as is shown in Figure 3.5b. Finally after the tagging process, a list of constraints representing the polygonal boundaries will be obtained, along with the ID information attached to indicate which polygons they should belong to (see Figure 3.5c and Figure 3.6c). Moreover, the triangulation itself will also be altered. Specifically, the faces in the CDT will be assigned with ID information as well so as to clarify which polygons they belong to. This would be useful when dealing with overlapping area, i.e. faces in the overlapping area will have multiple tags.

3.4 Snap rounding in the triangulation

In this section the core of the methodology will be introduced - snap rounding (SR) in the triangulation while also tracking the information of polygons. Put it simply, this requires two interrelated sub-steps: modify the constraints list obtained after the tagging stage and modify the triangulation (CDT). In the triangulation there are mainly two categories of cases to be snap rounded (see **Research Objectives**). They require different procedures, which will be explained in Section 3.4.1 and Section 3.4.2 respectively. Lastly and importantly, the significance of performing SR from the minimum case will be elaborated.

3.4.1 Snap rounding polygonal vertices

Identify the close polygonal vertices

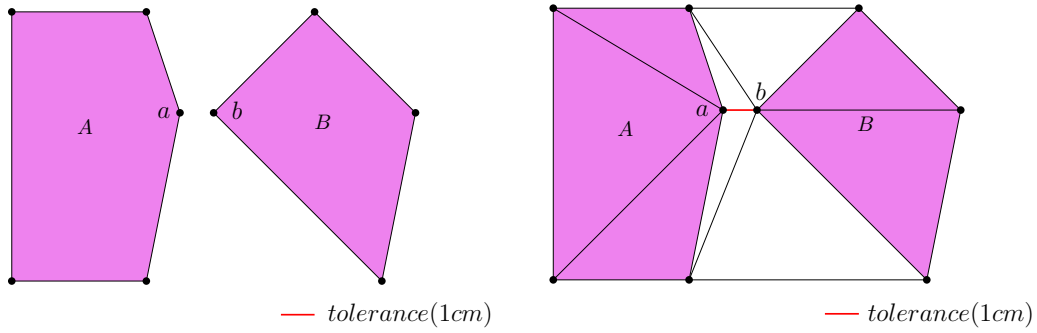
For an arrangement of polygons, the vertices can be very close to each other under a predefined tolerance. The very first step of snap rounding these vertices would be to clearly identify them. After having embedded the polygons into the CDT, all the vertices will be connected by edges hence the distance between any two vertices can be measured by computing the length of the edge connecting them. If the length of an edge is smaller or equal to the given tolerance, then the vertices connected by it would be considered as close polygonal vertices. It is noteworthy that when finding these close vertices we mainly focus on the edge length, thus it does not matter whether the connecting edge is constrained or not. However, in the following modification steps care needs to be taken when dealing with constrained edges and unconstrained edges.

Figure 3.10 shows an example. For the input polygons, after it has been embedded into the triangulation, the edges are traversed and the corresponding lengths are computed. The predefined tolerance is $1cm$, only one edge ab in the triangulation is either smaller or equal to it, as is shown in Figure 3.10b (highlighted in red). The vertex a and b are thus considered as close vertices, which require further modifications.

The details are described in Algorithm 3.4.

Modify the list of constraints

The next step after the identification of close polygonal vertices would be to modify them in order to eliminate the small gap between the vertices. The list of the constraints are firstly



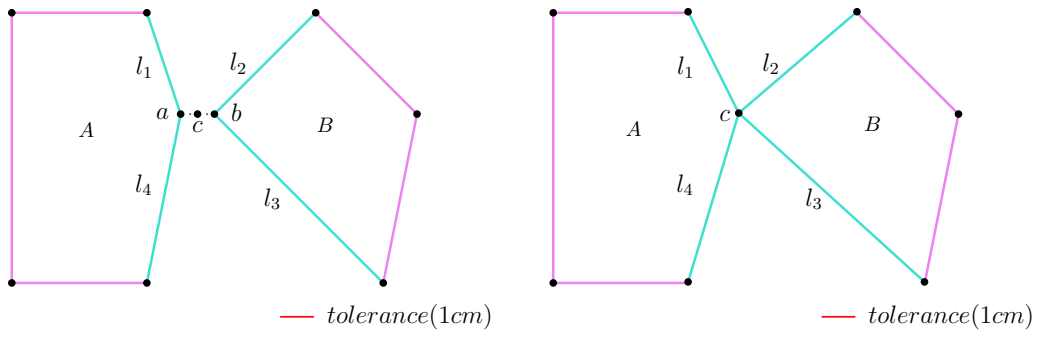
(a) (b)

Figure 3.10: Identify two close vertices. (a) Two input polygons. (b) Two polygons embedded in the triangulation. The length of edge ab is smaller than the tolerance ($1cm$), highlighted in red.

Algorithm 3.4: FIND CLOSE VERTICES (CDT, ε)

Input: A constrained Delaunay triangulation CDT, the snapping tolerance ε
Output: Close vertices (if found with the given tolerance)

```
for e in finite edges of CDT do
    get two endpoints of e;
    get the distance between the two endpoints d;
    if d ≤ ε then
        mark the two endpoints as close vertices;
        return close vertices;
return;
```



(a) (b)

Figure 3.11: Modify the list of constraints. (a) The constraints obtained after the tagging stage, which are stored in a list. The line segments highlighted in blue (l_1, l_2, l_3, l_4) are the constraints incident to vertex a and b respectively. c is the centroid of line segment ab, note that c does not belong to any polygon. (b) The constraints after the modification. The incident constraints l_1, l_2, l_3, l_4 are re-connected to vertex c.

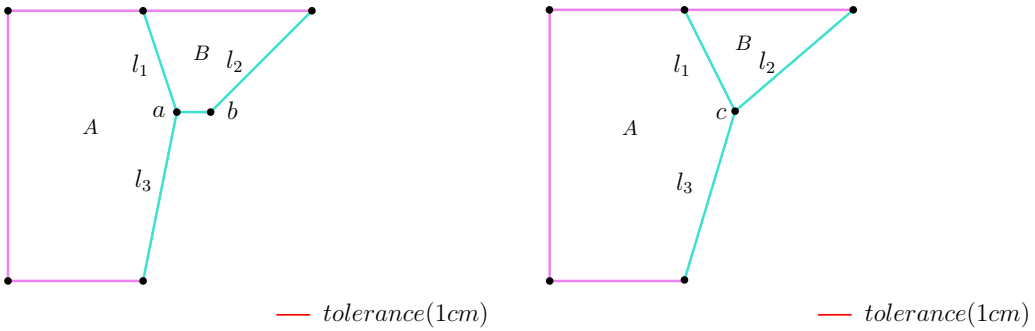


Figure 3.12: Modify the list of constraints (constrained edge). (a) The constraints obtained after the tagging stage, which are stored in a list. The line segments highlighted in blue (l_1, l_2, l_3, l_4, ab) are the constraints incident to vertex a and b respectively. (b) The constraints after the modification. The incident constraints l_1, l_2, l_3 are re-connected to vertex c.

modified in accordance with the polygon ID information. In other words, the boundaries of the polygons would be modified. See Figure 3.11 for an example. The line segments in Figure 3.11a represent the boundaries of two polygons. The blue ones (l_1, l_2, l_3, l_4) are the constraints incident to vertex a and b respectively. Since vertex a and b are too close to each other under the given tolerance, they need to be snapped as one vertex, which means a certain vertex will be used to replace them. The centroid of ab is a fair choice since it can avoid excessive shape changes of either polygon. The replacement will affect the constraints incident to a and b (l_1, l_2, l_3, l_4) but will have zero impact on other non-incident constraints. Therefore, the ending points of l_1, l_2, l_3, l_4 are modified from a, b to c, as is shown in Figure 3.11b. After the modification, the two close vertices are replaced by their centroid, and the original topology and geometry of the polygons are kept as much as possible.

It should be emphasized that, the edge ab in Figure 3.10b can be constrained in some cases, which indicates that ab is actually a boundary of a polygon (or a common boundary for adjacent polygons). In this case ab will be stored in the list of constraints, and it needs to be removed in the modification process. See Figure 3.12 for an illustration. In Figure 3.12a, the edge ab is one of the boundaries of polygon B. The constraints incident to vertex a and b are highlighted in blue, which includes l_1, l_2, l_3 and ab. When modifying the list, the constraint ab is firstly removed from the list, afterwards other incident constraints (l_1, l_2, l_3) are re-connected to the centroid c.

Modify the triangulation

Once the modifications of the list of constraints are completed, the triangulation (CDT) can be altered accordingly. The alteration towards the CDT is a must, as it is the basic data structure where the cases requiring snap rounding operations are found. Figure 3.13 illustrates the necessary steps in an example. The triangulation after the polygon A and B (see Figure 3.10a) are embedded is shown in Figure 3.13a. The blue line segments are the constrained edges in the triangulation, representing the boundaries of the polygons. Vertex a and b are close polygonal vertices and the edge connecting them is highlighted in red. The incident constrained edges of vertex a and b are denoted as l_1, l_2, l_3, l_4. Let us denote the ending points of l_1, l_2, l_3, l_4 (except for a and b) as *incident constrained vertices* (highlighted in purple), as they are incident to vertex a and b and connected by the constrained edges. For

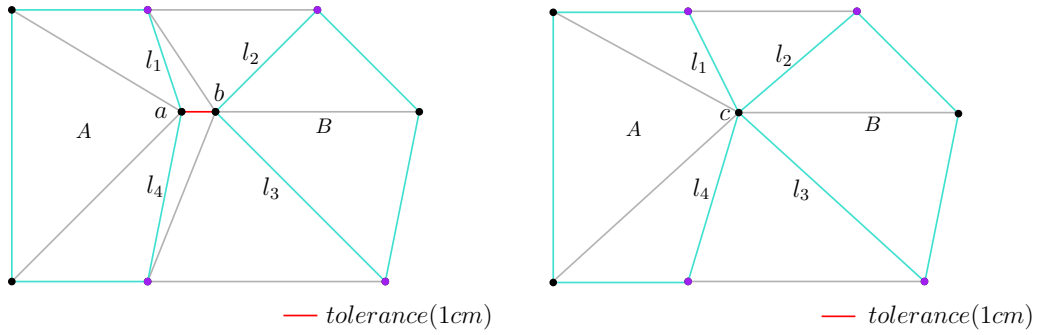


Figure 3.13: Modify the triangulation. (a) The triangulation after the polygon A and B (see Figure 3.10a) are embedded. The constrained edges are highlighted in blue, among which the l_1, l_2, l_3, l_4 represent the incident constrained edges of vertex a and b respectively. The edge ab is not constrained yet its length is smaller (or equal) comparing to the tolerance and highlighted in red. (b) The triangulation after the alteration. The incident constrained edges l_1, l_2, l_3, l_4 are re-connected to the centroid c (the centroid of ab).

the alteration, the edge ab is firstly removed in the triangulation (thus a and b are removed), this will also remove the constrained edges l_1, l_2, l_3, l_4. Afterwards the new constrained edges are re-introduced by connecting the *incident constrained vertices* to the centroid c, as is shown in Figure 3.13b. The newly constructed constrained edges are denoted as l_1, l_2, l_3, l_4 sequentially. This modification is local because of the characteristics of CDT [Chew, 1987], which ensures that the changes made will not affect the overall triangulation. This property makes the modification quite efficient, especially for a relatively large triangulation (e.g. containing more than 100,000 edges).

Resolve possible conflicts via merging process

The CGAL triangulation package [Boissonnat et al., 2000] automatically maintains the structure of the triangulation in response to the modifications, it also possesses the capability to resolve possible conflicts such as repeatedly inserting a constrained edge or a vertex. The package will omit the redundancies and keep the triangulation valid. However, the possible repetitions which may happen during the modification of the list of constraints will require manual processing.

Figure 3.14 shows a possible arrangement of two polygons. The vertex a and b are the close polygonal vertices since the distance between them is smaller than or equal to the given tolerance. As is described previously, they will be replaced by their centroid during the alteration. In Figure 3.15a the tagged constraints are shown. Constraint oa and ob are assigned the tag A and B respectively indicating which polygons they should belong to. In this case, the SR process will change the ending point of oa and ob from a, b to c, resulting in two constraints with different tags at the same location. The similar merging process mentioned in Section 3.3.3 is utilized to resolve this conflicts of repetition (see Algorithm 3.3). The result is depicted in Figure 3.15b, oc is the newly constructed constraint with tag A and B from both polygons.

Remove possibly dangling elements (optional)

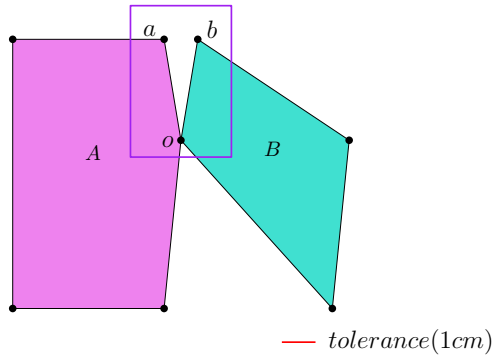
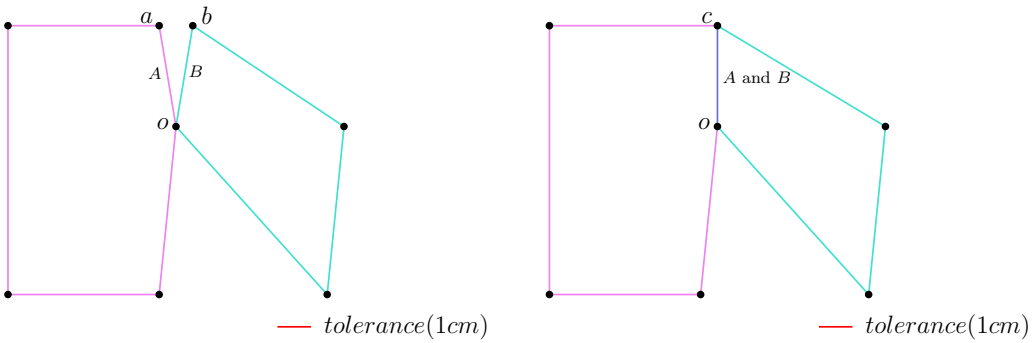
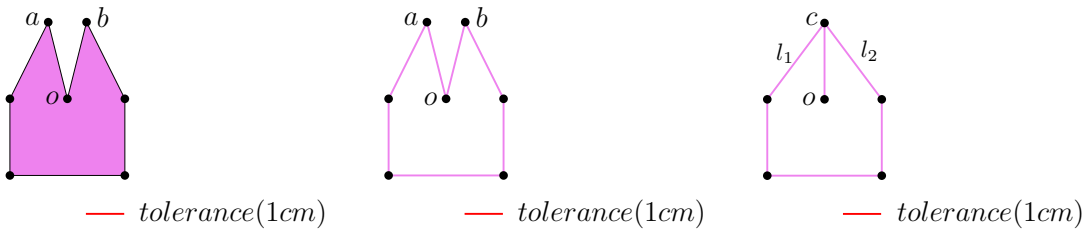


Figure 3.14: Two polygons which will possibly cause conflicts after snap rounding



(a) (b)

Figure 3.15: Possibly conflict constraints after snap rounding. (a) After the tagging stage, constraint *oa* and *ob* are constructed and assigned the tag (ID information) *A* and *B* respectively. (b) After the snap rounding, constraint *oc* is constructed and stored, having the tag *A* and *B* (*c* is the centroid of *ab*).



(a) Input polygon. (b) Tagged constraints. (c) After modification.

Figure 3.16: Possible dangles during the modification of the list of constraints.

So far most of the content of performing SR on polygonal vertices have been discussed. There is one more thing that is worth mentioning: the possible presence of dangling elements and the related cascading effects. See Figure 3.16 for a straightforward illustration. In Figure 3.16a, vertex a and b are close polygonal vertices under the predefined tolerance. After the tagging stage a list of constraints can be obtained, as is shown in Figure 3.16b. Care needs to be taken when the list of constraints are being modified, as in this case the alteration will create a dangling constraint oa (Figure 3.16c). Note that due to the dangling constraint the vertex o will also become a dangling vertex, which can be very close to line segment l_1 or l_2, resulting in new cases (close polygonal vertex and boundaries). In addition, the distance between vertex o and c (the length of oc in Figure 3.16c) can also be smaller than (or equal to) the threshold, which suggests that o and c become new close polygonal vertices. Therefore, they need to be rounded again. The cascading effects, in our context, stand for the fact that *the SR operation performed on one case would possibly create new cases that require snap rounding modification again*.

The cascading effects can be reduced or avoided by removing the dangling line segments and vertices. Technically speaking, the dangling line segment can be firstly identified in the list by a traversal and then deleted from the list during the snap rounding process. Locating dangling vertex can be slightly more difficult. Observing from Figure 3.16c, a conclusion can be drawn that non-dangling vertices are incident to at least two constraints while the dangling vertex is only incident to one constraint. This can be used to find the dangling vertex. It should be pointed out that in the methodology a list is being utilized to store all constraints, therefore in practice locating and removing the dangling elements often requires a traversal operation (or multiple traversals). The traversal operation will become very expensive hence consume a significant quantity of time resources when the number of constraints in the list is relatively large (e.g. more than 100,000). Considering the cost and the dangling elements are not very common (depending on the polygon shape), the removing process is made optional. For small or middle dataset, enabling the removing operation would possibly yield a result with better quality, while for large or massive dataset, removing process is recommended to deactivate as the time cost would be too expensive to accept for most users.

Another issue of dangling elements is that the dangling line segments (constraints) will make a ring of a polygon invalid. As is demonstrated in Figure 3.16c, the constraints (highlighted in violet) will not form a valid and enclosed polygon (Figure 3.16b shows an example of an enclosed ring which will form a valid polygon). This issue can be automatically handled in the polygon reconstruction stage. That being said, the dangling elements caused by the SR will not have negative impacts on the polygon reconstruction.

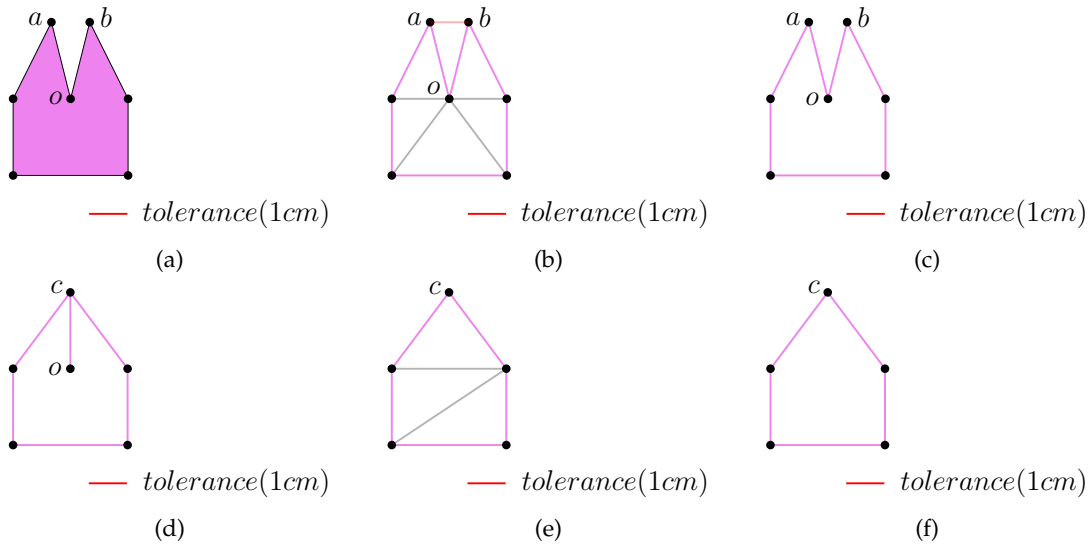


Figure 3.17: Summary of snap rounding polygonal vertices. (a) Input polygon. (b) The polygon is embedded into a triangulation (CDT), where close polygonal vertices (a, b) are identified under the given tolerance. (c) Tagged constraints (boundaries). (d) Modify the list of constraints. (e) Modify the triangulation (CDT). (f) Removing dangling elements.

Summarize

Based on the content above, the main steps of *snap rounding polygonal vertices* can be summarized as below:

- Identify the cases (close polygonal vertices) under a certain threshold (Figure 3.17a, Figure 3.17b);
- Modify the list of constraints: remove close vertices, use the centroids instead, resolve possible conflicts of repetition when updating constraints, remove possibly dangling elements; the aim of this process is to keep the shape of polygons (represented by the boundaries) up to date (Figure 3.17c, Figure 3.17d, Figure 3.17f);
- Modify the triangulation (CDT): remove the edges connecting the close vertices and re-introduce the constraints between the centroids and the *incident constrained vertices* (Figure 3.17e);

An illustration of main stpes is provided in Figure 3.17 (including removing possibly dangling elements). The Figure 3.17a shows an example input polygon, the ideal output of this stage is depicted in Figure 3.17f.

3.4.2 Snap rounding polygonal vertex and boundaries

Identify the close polygonal vertex and boundaries

In an arrangement of polygons, there are not only close polygonal vertices described in Section 3.4.1 but also close polygonal vertex and boundaries. If the distance between a vertex and a boundary of a polygon is smaller than (or equal to) a predefined tolerance,

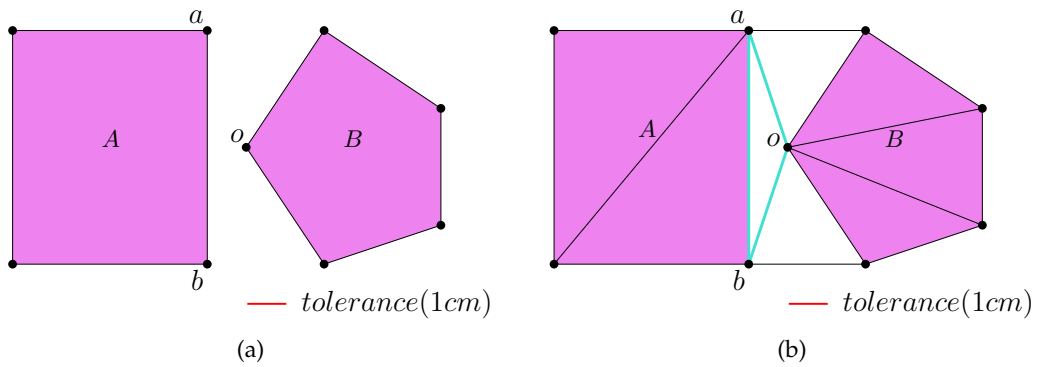


Figure 3.18: An arrangement of two polygons with a vertex being close to a boundary. (a) Two polygons A and B. Vertex o of polygon B is close to the boundary ab of polygon A. (b) The triangulation after two polygons are embedded into a triangulation. $\triangle$ abo is the incident triangle of vertex o and boundary ab.

then this vertex and the boundary need to be snap rounded. The identification of such case would rely on the constructed triangulation. Observing from Figure 3.18, if a polygonal vertex is very close to a polygonal boundary under a certain tolerance, this vertex is called a *capture vertex* and the incident triangle ($\triangle$ abo in Figure 3.18b) is a *possibly sliver triangle*, as the triangle is rather skinny comparing to other well-shaped triangles in the CDT.

Now let us take a closer look at $\triangle$ abo. As is shown in Figure 3.19a, the distance between vertex o and boundary ab can be represented as the length of oo' (denoted as h). Boundary ab is expressed as a constrained edge in the triangulation, the other two edges oa and ob may be or not be constrained. Figure 3.19b illustrates another possible shape of the triangle $\triangle$ abo. The length of oo' is smaller than (or equal to) the tolerance and ab is also constrained. Unlike the previous situation, the projection of vertex o on ab (o') is not within the bounds of line segment ab as it falls outside the range of ab, which makes $\triangle$ abo not a *possibly sliver triangle* to be used in our snap rounding operation. In other words, a triangle with a shape similar to Figure 3.19b will not be considered a valid *possibly sliver triangle* in our case. The logic behind this is that if the projection of a vertex on the corresponding close boundary is outside the range, it is of no reason that this vertex should be re-connected to the boundary.

Modify the list of constraints

Similarly, following the identification stage, the list of constraints would be modified first in order to remove the small gaps between vertices and boundaries. Refer to Figure 3.20 for an illustrative example. The constraints (stored in a list) obtained from the tagging process are illustrated in . The *capture vertex* o is very close to the boundary ab (highlighted in blue). With the aim of snapping o and ab together, new constraints oa and ob are created and added into the list. Afterwards ab is removed. It is important to know that constraint oa and ob are derived from the triangulation (see Figure 3.18b). By replacing ab with oa and ob, vertex o and boundary ab are touching hence the small gap is removed. One can argue that such modification would cause unintended distortions (i.e. edge ab is stretched to edge aob) and maybe it is a better idea to directly move o onto ab. However, recall that all of the constraints are constructed from the triangulation during the tagging stage hence removing constraints and inserting new constraints would require corresponding modifica-

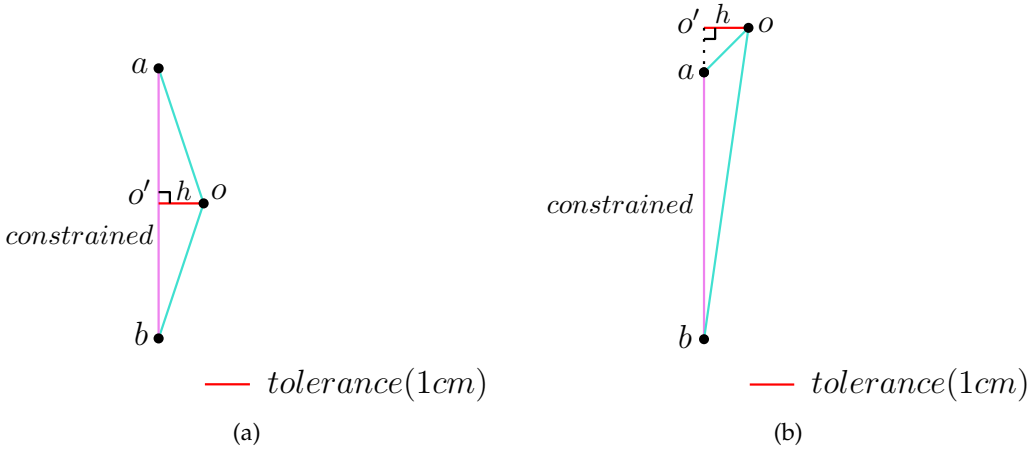
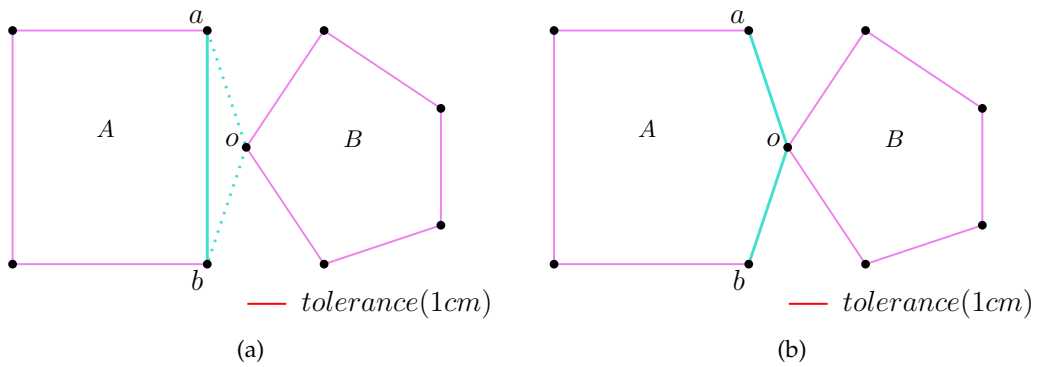


Figure 3.19: A possibly sliver triangle. (a) The projection of vertex o on ab (o') is between a and b. (b) The projection of vertex o on ab (o') is outside the bounds of line segment ab.

tions in the triangulation (CDT) as well. Deletions and insertions of the constraints are less robust due to the changes to be made would affect the structure of the CDT significantly and might introduce invalidities such as constraint intersections. Albeit the CGAL triangulations package [Boissonnat et al., 2000] has the capability to handle the possible errors, such automated modifications may cause us to lose dynamic tracking of polygon shapes during the SR process.

Modify the triangulation

After having completed the modifications of the constraints, the CDT, where all elements are embedded, is ready to be altered correspondingly. Considering the processing performed during rounding polygonal vertices (Section 3.4.1), the triangulation would require no explicit insertions but only changes of constrained status of incident edges regarding rounding polygonal vertices and boundaries. An example is shown in Figure 3.21. Upon completion of having embedded the polygons, the structure of the triangulation is depicted in Figure 3.21a, each blue line segment is indicative of a constrained edge and vertex o is the *capture vertex* with the given tolerance (being very close to boundary ab). Non-constrained edge oa and ob are made constrained after the modification, and constraint ab is removed (note that only constraint is removed, vertex a and b are still in the triangulation). Since the constraint between a and b no longer exists, the CDT automatically updates its structure locally so as to maintain its Delaunay property (recall that the Delaunay property means the triangles are organized as well-shaped as possible in a CDT). The biggest advantage of this approach is that it minimizes the necessary changes that need to be done of the triangulation. Such behaviour would make the modifying operation as robust as possible. Previously it is mentioned that the *capture vertex* o could also be snapped onto the boundary ab, however this is rather troublesome as it requires new insertions, including the insertion of a new vertex o' on the edge ab (see Figure 3.22) and insertions of new constraints connecting o' and the incident vertices of the original *capture vertex* ($o'c, o'd$ in Figure 3.22). Insertions are typically expensive in a CDT since the triangulation needs to determine whether the incoming vertex has already existed or not, this would probably result in a global traverse in some cases. Additionally, there is a possibility that the insertion of a constraint would cause unintended consequences such as intersecting with other constraints (e.g. in Figure 3.22 newly-added



(a) (b)

Figure 3.20: Modify the list of constraints for close vertices and boundaries. (a) The constraints (representing polygon boundaries) obtained after the tagging stage. Line segment *ab* is highlighted in blue to indicate it is very close to a polygonal vertex (*o*). Note that there are two line segments *oa* and *ob* which are depicted in blue dots, they have not been added to the list of constraints yet. Vertex *o* is the *capture vertex* and triangle $\triangle oba$ is its incident triangle. (b) The constraints after the modification. Original constraint *ab* is removed, constraint *oa* and *ob* are introduced.

constraint $o'c$ and $o'd$ may interfere with other constraints if the triangulation has a very complex topology). Having these considerations, this thesis eventually chooses the method depicted in Figure 3.21 to perform necessary rounding operations.

Sliver triangle

In the identification stage it has been described and demonstrated the concept of a *possibly sliver triangle*, based on which the *capture vertex* is located and snapped. However, in practice it has been found that the rounding schema would result in infinite loops for certain *possibly sliver triangles*. See Figure 3.23 for an example. After constructing the CDT, an excerpt of the triangulation is depicted in Figure 3.23a ($\triangle pqr$), with vertex *r* being very close to the constrained edge (boundary) *pq*, *r* is the *capture vertex*. Subsequently the rounding schema is applied, edge *pr* and *qr* are ensured to be constrained and edge *pq* are made non-constrained (shown in Figure 3.23b). Note that after removing constraint *pq*, the edge *pq* still exists. Be advised that *qr* is perpendicular to *pr*, the projection of *q* on *pr* is *r*, and the distance from *q* to *pr* is the length of *qr*. If the length of *qr* is smaller than the given tolerance then the vertex *q* will be considered close to the constrained edge *pr* and be snapped again, resulting in the arrangement illustrated in Figure 3.23c. The vertex *r*, once again, becomes the *capture vertex* as the distance between *r* and constrained edge *pq* is smaller than the tolerance. Rounding operation of (c) will lead to the result shown in Figure 3.23d, which is exactly the same as Figure 3.23b. It can be easily inferred that continuing SR will result in infinite loops of the program.

Considering the situation above, the principle of locating close polygonal vertex and boundaries needs to be refined. In order to clearly elaborate the refinement, several basic concepts need to be demonstrated as follow:

- *sliver base edge*: In a triangle, if the distance between a vertex and an edge is less than a given tolerance, the edge is called a *sliver base edge* relative to this vertex (e.g. in Figure 3.23a, consider vertex *r*, its *sliver base edge* is edge *pq*; consider vertex *q*, its *sliver base edge* is edge *pr*);

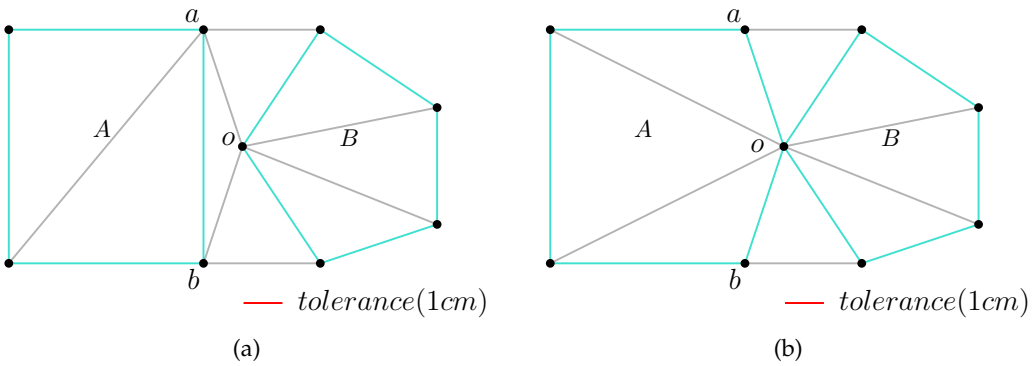


Figure 3.21: Modify the triangulation for close vertices and boundaries. (a) The triangulation with polygon A and B embedded. The constrained edges are highlighted in blue. Vertex o is considered as the *capture vertex* given the tolerance (being very close to boundary ab). Triangle oab is a *possibly sliver triangle*. (b) The triangulation after the alteration. Edge oa and ob are ensured as constrained. Constrained edge ab is marked as non-constrained. The triangulation within polygon A changed due to the new constraints.

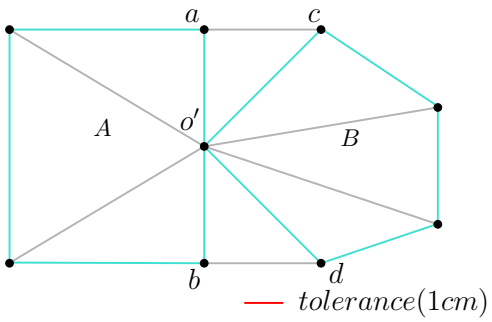


Figure 3.22: Another possible way of snap rounding polygonal vertex and boundaries.

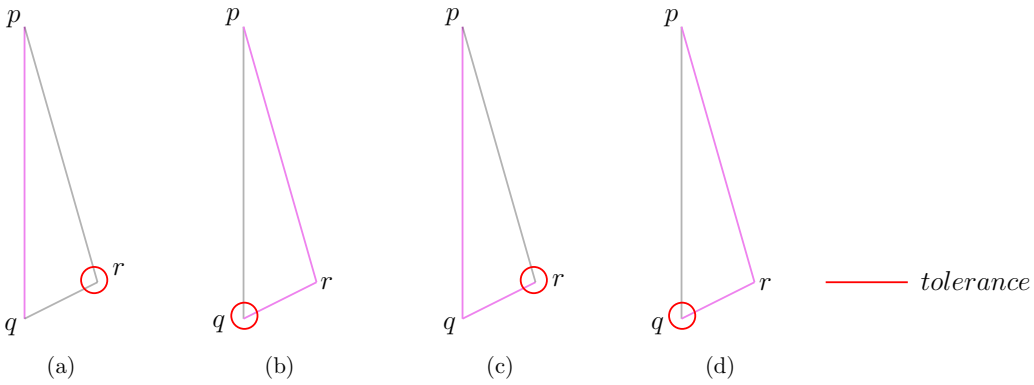
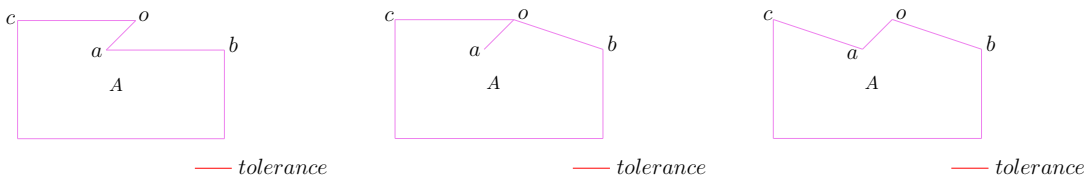


Figure 3.23: Infinite loops when snap rounding a *possibly sliver triangle*. The *capture vertex* (which will be snapped) is highlighted in red circle.



(a) (b) (c)

Figure 3.24: Remove dangling elements when rounding close vertex and boundaries. (a) The input polygon. (b) Polygon after snap rounding polygonal vertex and boundaries.

- *constrained sliver base edge*: In a triangle, if a *sliver base edge* is constrained, then it is called a *constrained sliver base edge* (e.g. in Figure 3.23a, consider vertex r, its *constrained sliver base edge* is edge pq;)

Having these in mind, a triangle that satisfies the following requirements is defined as a *sliver triangle*:

- The triangle must contain only one *constrained sliver base edge*.
- Except for the *constrained sliver base edge*, the other two edges must not be *sliver base edges* (they can be constrained or non-constrained).

With the introduction and utilization of *sliver triangle*, the infinite loops caused by a *possibly sliver triangle* can now be resolved. Problematic cases such as the one in Figure 3.23 are omitted.

Remove possibly dangling elements (optional)

Rounding polygonal vertex and boundaries can also create dangling elements in some cases. See Figure 3.24 for an example. Polygon A has a so-called "z" shape ($c - o - a - b$), with vertex o being close to boundary ab under the predefined tolerance. In accordance with the rounding schema, performing snap rounding on polygon A would result in the shape depicted in Figure 3.24b. As is shown in the figure, in the rounded counterpart segment oa becomes a dangling line segment and vertex a becomes a dangling vertex. The problem lies in the fact that the distance between the dangling vertex a and the boundary oc is smaller than the tolerance as well, hence a will be snapped again, leading to the final shape in Figure 3.24c. If the dangling vertices and segments are firstly detected and processed before the next snap rounding, the final rounded result would be the polygon shown in Figure 3.24b but without segment oa. Generally speaking, handling dangling elements will probably yield a better result since the polygon distortions are smaller. Besides, the dangling elements are created as a kind of side effect during the SR process, further rounding operations should not be applied on such intermediate objects.

As outlined earlier, even though removing dangling elements is capable of improving the quality of rounded results and reduce cascading effects, it is itself a relatively expensive operation. Therefore, there is always a trade-off between the results with higher quality and the overall resource costs of the program.

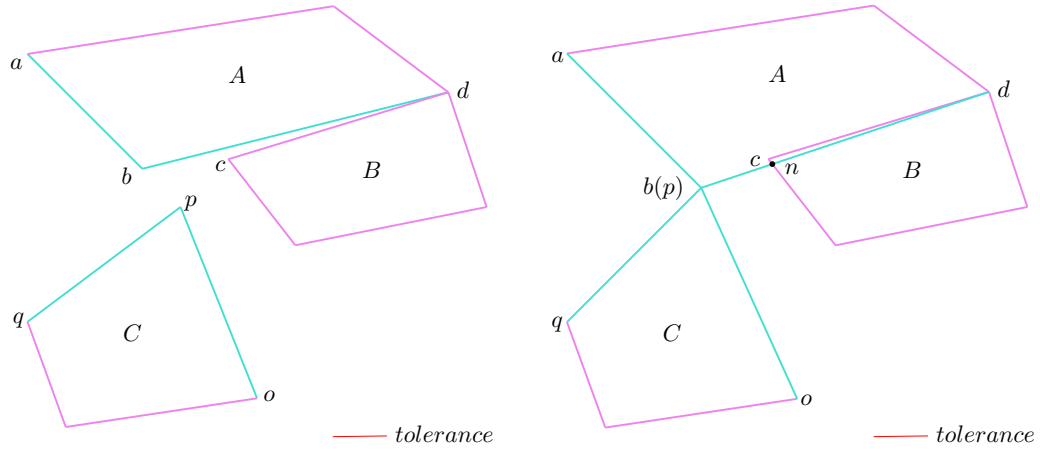


Figure 3.25: Apply SR on a set of polygons. (a) Input polygons. Boundaries that will be modified are highlighted in blue. (b) Polygons after SR. Boundary bd intersects polygon B at vertex n after first rounding operation.

3.4.3 Snap rounding from the minimum case

In real world dataset, there can be multiple cases requiring SR operations with a certain tolerance. Different rounding sequences can lead to different results. Figure 3.25 depicts an example. Observing from the left figure, the distance between vertex b (of polygon A) and p (of polygon C) is smaller than the given tolerance thus they are considered as close polygonal vertices which need to be snap rounded. Note that the distance between vertex c (of polygon B) and boundary bd is even smaller than that between vertex b and p. If vertex b and p are snapped first, the revised constraint $b(p) - d$ would intersect polygon B and create a new vertex n at the intersection, as is shown in Figure 3.25b. The length of segment cn is smaller than the tolerance hence a new rounding case is created. This is a representative example of the aforementioned concept *cascading effects*. It is noteworthy that the intersection also causes the overlap (triangle $\triangle cnd$) between polygon A and B, which makes the arrangement more complicated to handle.

A realistic way to avoid such issue is to ensure that the case with the smallest distance under a certain threshold (called a *minimum case*) is rounded first. As is illustrated in Figure 3.26, vertex c and boundary bd are snapped together prior to processing vertex b and p. The resulting arrangement is shown in Figure 3.26b. It is apparent that in this arrangement such manner would not create any intersections nor overlaps.

3.5 Reconstruct Polygons From the List of Constraints

Once the SR process is finished, the list of constraints (representing the polygonal boundaries) are used to reconstruct the polygons. As this final step mainly serves as an auxiliary role to help reconstruct polygons, using existing and mature algorithms might be an efficient

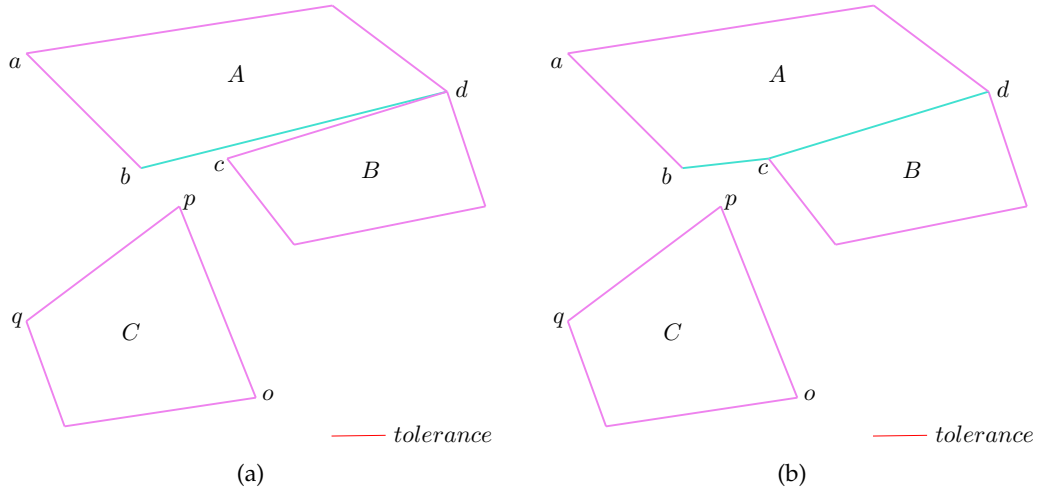


Figure 3.26: Apply SR on a set of polygons, starts from the *minimum case*. (a) Input polygons. Boundaries that will be modified are highlighted in blue. (b) Polygons after SR. Vertex c and boundary bd are considered as the *minimum case*, hence they are snapped first. There are no intersections nor overlaps after the first snapping process.

solution. The GEOS (Geometry Engine, Open Source) is a C/C++ library that provides abundant functions to perform geometric and spatial computations [GEOS contributors, 2021]. It has a `Polygonizer` class, which implemented several functions to build 2-dimensional polygons from a set of line segments. The `Polygonzier` roughly proceeds as follows. Firstly it constructs a graph representation using the input line segments. Each line segment is considered as an edge in the graph and each vertex is represented as a node. After adding all nodes and edges, it then traverses the graph, starting from a node or an edge, to analyse the connectivity of the graph in order to identify potential closed rings. To achieve this, the `Polygonzier` traverses adjacent line segments and checks whether there are possible connections of the segments to form a closed ring. Different criteria may be taken into consideration in this process, such as if there exists shared endpoints or important geometric relationships like intersections or touching. Once a potential ring is found and formed, it then undergoes validation to ensure the geometric and topological correctness of the ring. For instance, a valid ring must not self-intersect. After having validated the formed rings, the `Polygonizer` constructs polygon geometries using the formed rings. The constructed polygons are valid, but they may contain interior holes. Care needs to be taken when exporting these polygons to the output file. An example of polygon reconstruction is shown in Figure 3.27. Polygon A in Figure 3.27a has two inner holes: h_a and h_b. The `Polygonizer` firstly builds a graph based on the input line segments, then it processes the constructed graph to detect the intersection points amongst line segments and splits them respectively. The resulting segments are finally connected to form enclosed rings. The reconstructed result is shown in Figure 3.27b.

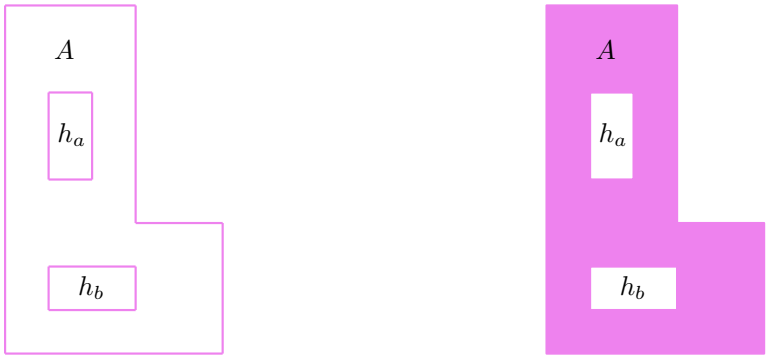


Figure 3.27: An example of using polygonizer to polygonize a set of line segments. (a) A set of line segments. (b) The resulting polygon of `Polygonizer` containing two interior holes.

4 Implementation

In order to investigate the applicability of the proposed method, a simple prototype (called *snapoly*, meaning "snap polygons") has been developed using C++ as the programming language.

4.1 Data Structures

In common practice, a program usually becomes increasingly complex as the development progresses. Appropriate and intuitive data structures would help to clarify and better organize the procedures of the implementation. It is worth noting that simplicity does not necessarily mean good efficiency. Sometimes high performance requires sophisticated design. Regarding the prototype of this thesis, the number one rule should be that it is designed and realised as straightforward as possible, meanwhile with the potential of further development and optimization to achieve better performance.

Several data structures have been designed for the prototype, they will be described as follow:

Enhanced_constrained_delaunay_triangulation_2

This class is inherited from the `CGAL::Constrained_Delaunay_triangulation_2` class, with some additional functions which are required during the SR process. Using an derived class from CGAL has certain benefits, i.e. accessing the CGAL internal functions and having the flexibility to add and customize own functions. This class is mainly used for constructing and maintaining the triangulation.

Constraint

Recall that in the methodology the polygonal boundaries need to be stored and dynamically updated. Boundaries can be considered as line segments with ID information (of the polygons which they belong to) attached. `Constraint` is the class to store the boundaries, it contains two points indicating the endpoints of a line segment and a container to store possible ID information. Note that for one constraint (boundary), it principally can have multiple IDs (e.g. common boundary). Additionally some basic operators are also provided.

CDTPolygon

This is a class similar to `CGAL::Polygon_with_holes_2` class which is used to store a polygon in 2-dimensional space with (possible) interior holes. The class members include a `CGAL::Polygon_2` object representing the exterior ring of a polygon, a container with the type `CGAL::Polygon_2` to store possible holes (the container will have size zero if no holes are present) and a string to store the ID information (one polygon can only have one ID in a

valid dataset). The `CDTPolygon` class also provides elementary functions to perform calculations related to polygons such as obtaining the number of interior holes, querying if a point lies inside or outside of a polygon, calculating the area, etc.

Snap_rounding_2

This class encapsulates essential objects and functions to perform SR on polygons. Since implementing the whole procedure of the method would result in many lines of code, it is a good practice to make functions organized according to different purposes. This is relatively a big class comparing to others and has more class members, they are arranged in the following order (note that the members have a common prefix `m` to indicate they are class members not ordinary or temporary variables):

- `m_tolerance`: The tolerance of SR operation. Recall that elements (usually vertices and boundaries) with a distance less than this value would be modified and captured together.
- `m_squared_tolerance`: The squared value of `m_tolerance`. This value is actually used to compare with the (squared) distance between vertices or between a vertex and a boundary. The reason behind this is that multiplication is typically faster than division in modern computers and the distance between elements is firstly computed as a squared value. Consider that taking square root can be a more complex and expensive operation, using squared values is a better option.
- `m_et`: An instance of a triangulation, where all input polygons are embedded. The triangulation would be updated automatically as the snap rounding operation proceeds.
- `m_constraintsWithInfo`: A container with the type `constraint`. The boundaries of polygons are stored in this container. They are updated manually and dynamically during the snap rounding process. With the utilization of C++, various containers can be used, such as `std::list`, `std::unordered_set` and `std::unordered_map`.
- `m_polygons`: A container with the type `CDTPolygon`, storing the input polygons.
- `m_result_polygons`: A container with the type `CDTPolygon`, storing the reconstructed polygons after the snap rounding.

4.2 Prototype

With the objective of developing a prototype with the potential of achieving high efficiency, C++ was selected as the main programming language. C++ is a compiled language, which means the code gets compiled into machine code before it is executed. For modern computers, the compiled code can be directly run without the need to be interpreted during runtime thus programs written in C++ can usually achieve good performance. Another important factor is that third-party libraries: GDAL and CGAL are frequently used in the implementation. They both have C++ interfaces (GDAL provides C and C++ APIs and CGAL is originally written in C++) thus it is easier to use and integrate them in a C++ environment.

For the consideration of convenient management, header files and source files are placed separately. Header files ordinarily contain class & function declarations, but it may also include a few inline function definitions. Most of the function definitions and implementations are in source files. Furthermore, in order to accelerate the compilation stage, precompiled

header file (`.pch`) is used. It usually contains the preprocessed output of commonly used header files. Therefore when the source code gets compiled, the compiler can sorely use the precompiled files instead of parsing the common header files over and over again. This can save a significant amount of time especially during the debugging stage.

The prototype is available at https://github.com/zfengyan/snapoly, under a GNU General Public License (v3.0). To view a copy of this license, visit `https://www.gnu.org/licenses/gpl-3.0.html`.

4.3 Engineering Decisions

It is critical to be aware that different choices of implementation tactics would lead to different output, even in some cases, determine whether the program runs successfully or fails. The efficiency of the program also depends on the specific decisions. In this section, several important details and selections are presented and discussed. Note that these decisions are not necessarily the best options to achieve potentially highest performance.

Use std::list as the container to store constraints

As previously described, the boundaries of polygons (namely, constraints) are stored in a container and will be dynamically updated during the snap rounding process. In the chapter **Methodology**, it is demonstrated that the `constraints` would be regularly modified, including removing and inserting operations. Using `std::list` is a relatively good practice to provide fast insertions and erasures for stored elements regardless of the locations in the list. The disadvantage is that it is using non-contiguous memory thus it would generally consume more spaces compared to containers using contiguous memory such as `std::vector`.

Use $1e-8$ as the threshold value when comparing two doubles

The `double` type in C++ can not be compared directly if one wants to ensure the correctness of the comparing result. This is due to the precision limits of floating-point arithmetic. To make a straightforward and simple illustration, assume two `doubles`: 10000.000000000001 and 10000.000000000002, conceptually the latter is greater than the former while inside the memory they could be both represented as 1.0000000000×10^4 (just for an example, the actual precision is usually higher) using floating-point arithmetic, which indicates they are considered the same. Therefore, a threshold is often used to "safely" compare the value of two doubles especially in the scenarios where inaccurate or randomized comparison result can lead to invalid output. Choosing a proper value of the threshold can be tricky sometimes. If a threshold is too small, such as $1e-15$, it may take irrelevant differences (caused by rounding errors of floating-point numbers) into consideration, leading to unnecessary comparisons. On the other hand, if a threshold is too large such as $1e-3$, it may miss some differences between `double` values thus yield incorrect comparison results. According to some experiment results conducted during implementing the prototype, it is discovered that the difference between two "same" variables with `double` type usually ranges from $1e-6$ to $1e-12$, depending on the magnitude of the values. Based on the testing results of different datasets, $1e-8$ is eventually used as the value of threshold. It should be emphasized that the appropriate threshold value should be chosen in accordance with the application context and decided by the users. There is no rigid regulation or standard regarding this.

5 Results and Analysis

5.1 Datasets

In order to verify if the developed prototype meets the requirements, diverse datasets that contain multiple polygons have been collected and tested, see Table 5.1. Three different categories of datasets are primarily selected. The first category, termed *Normal*, represents ordinary towns. Datasets of this type usually incorporate both densely and sparsely built areas, as well as natural landscapes, parks, and other possible features. The second representative category is *Dense urban area*, which typically represents the bustling city center regions. The building density in these regions is usually high and the buildings are usually arranged tightly. Datasets of such category would possibly contain more SR cases (close vertices and boundaries) with a certain tolerance. The third type is *Island*, where the arrangement of buildings is usually irregular and spaced apart from each other. Note however, the difference between the category *Island* and *Normal* is not significant.

All of the datasets are downloaded from GeoFabrik [GeoFabrik, 2023], which provides free downloads of open-source geospatial data from OpenStreetMap (OSM) [Bennett, 2010]. The downloaded data typically contains many different `ESRI Shapefiles` indicating different types of elements. An example of a downloaded dataset is depicted in Figure 5.1, the whole dataset covers the main area of *Andorra*, the figure shows an excerpt that contains buildings (in grey), landuse (in brown), water (in blue, including water body and waterways), railways and roads (black line segments). In this thesis the polygons (buildings in the original dataset) draw more attention than other entities, thus the downloaded dataset is firstly preprocessed so as to filter out polygons. This can be easily achieved in QGIS [QGIS Development Team, 2009] by simple selection and exportation. Another consideration to keep in mind is that the downloaded dataset mostly uses `EPSG:4326` as its CRS. `EPSG:4326` uses the `WGS 84` datum and does not use any cartesian coordinate systems (with x, y and z axes), instead, it utilizes a spherical coordinate system to describe the locations on the surface of the Earth. Therefore, the coordinates are initially expressed as latitude and longitude.

Using latitude and longitude coordinates to perform geometric computations would require extra steps, namely, to utilize the great circle distance [Carter, 2002]. Therefore this is

	Abbreviation	Number of polygons	Type
Den Haag Central	DHC	19878	Dense urban area
Faroe Islands	Faroes	30926	Island area
Delft	Delft	38376	Normal
Freiburg	Freiburg	53723	Normal
Rotterdam Central	RCD	127795	Dense urban area

Table 5.1: Testing datasets for the prototype.

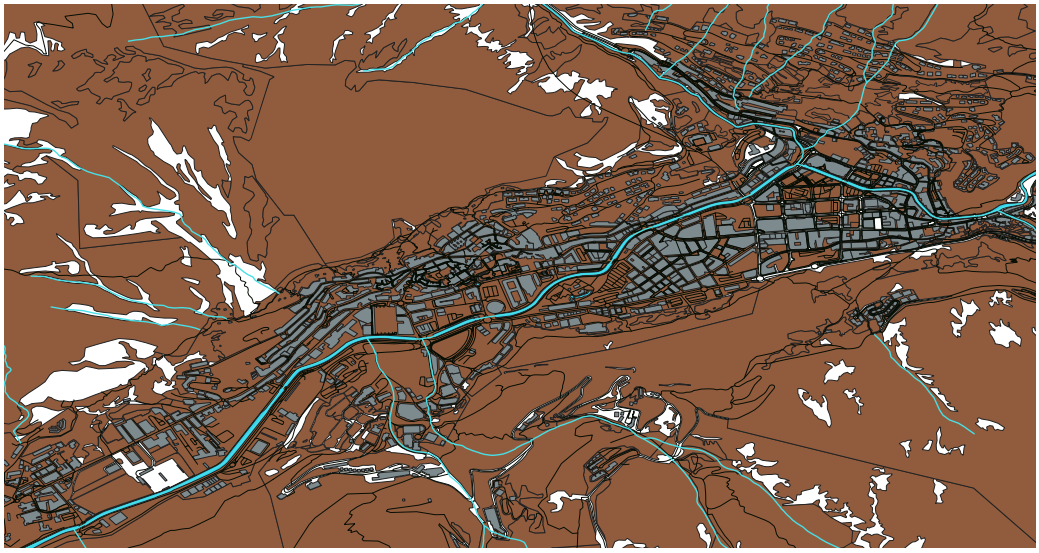


Figure 5.1: An excerpt of downloaded Andorra dataset. Buildings are depicted in grey, landuse is shown in brown, water body and waterways are in blue, the black straight lines represent the roads and railways.

not the most suitable option when geometric computations (e.g. calculating the distance between two points) are involved. A pragmatic way to avoid this is to use a projected CRS, which converts the latitude and longitude into x and y coordinates. Keeping in mind that CRS would cause distortions thus it needs to be carefully selected. Fortunately EPSG (`https://epsg.io/`) provides a large number of CRS to be applied on different regions. Users can use various CRS according to their specific needs. The polygons filtered and projected are shown in Figure 5.2.

5.2 Evaluation

5.2.1 Symmetrical difference

Symmetrical difference is a common concept in the field of GIS, it typically computes the geometric intersections of the input features. The area that do not overlap will be returned as the result. This would enable us to visually identify the differences (namely, the changes of the shape of the polygon) before and after the SR process. An illustration is shown in Figure 5.5. Figure 5.5a shows the two polygons (with a small gap) in the dataset, and the rounded result (with the small gap eliminated) is depicted in Figure 5.5b. The differences before and after SR are presented in Figure 5.5c. In the practice, datasets are usually large and often comprise a large number of polygons, hence the differences (or distortions) are not straightforward to observe. As is shown in Figure 5.3. The Delft dataset and its rounded counterpart look similar from a normal scale level. To gain insights into the differences and verify if the small gaps have been corrected, symmetrical difference is computed and the result is illustrated in Figure 5.4. According to the result, it becomes possible to backtrack the original and the rounded polygon arrangement for the purpose of verification. For instance,

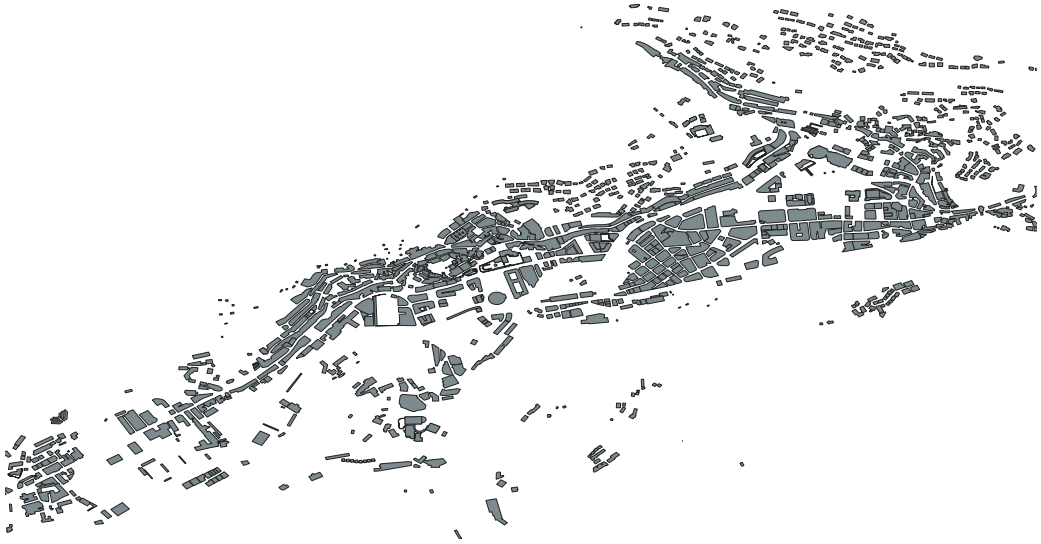


Figure 5.2: An excerpt of downloaded Andorra dataset with polygons filtered. It only contains buildings which are depicted in grey.

(a) Input. (b) Rounded result.

Figure 5.3: An excerpt of Delft region.

by following the enlarged purple circle in Figure 5.4, once can observe the original polygon arrangement shown in Figure 5.5a and the snap rounded result in Figure 5.5b. Therefore it can be verified that the small gaps between these two polygons have been successfully corrected. This approach allows for a straightforward visual confirmation of the results achieved through the process of SR.

5.2.2 Area difference

Symmetrical difference would permit to intuitively observe and identify the differences between the input dataset and the rounded counterpart in QGIS. However it can not be easily quantified. It should be emphasized that, quantitative data comparison holds significant importance in evaluating results, particularly for large datasets. Regarding the input and output, such quantitative indicators can directly reveal the extent of distortions before and after the rounding operation.

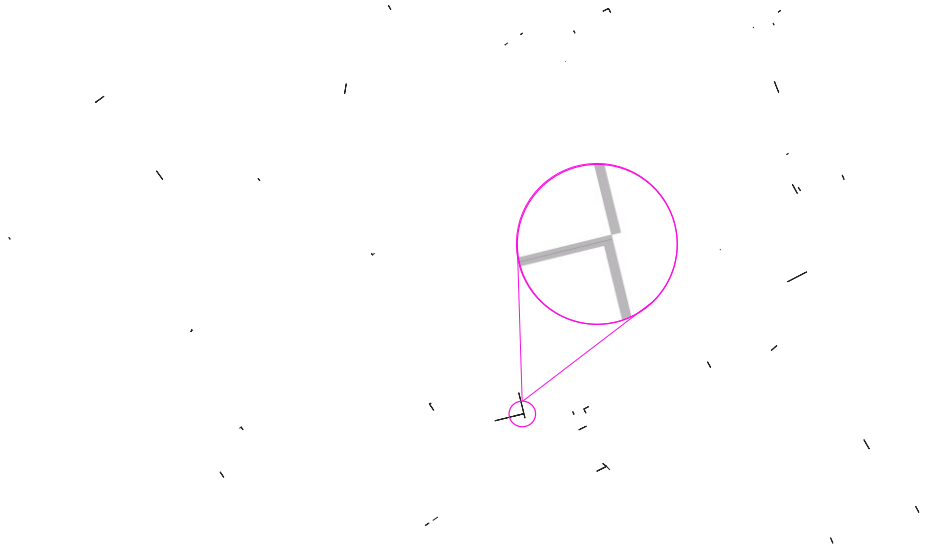


Figure 5.4: The symmetrical difference of the excerpt of Delft region (shown in Figure 5.3).

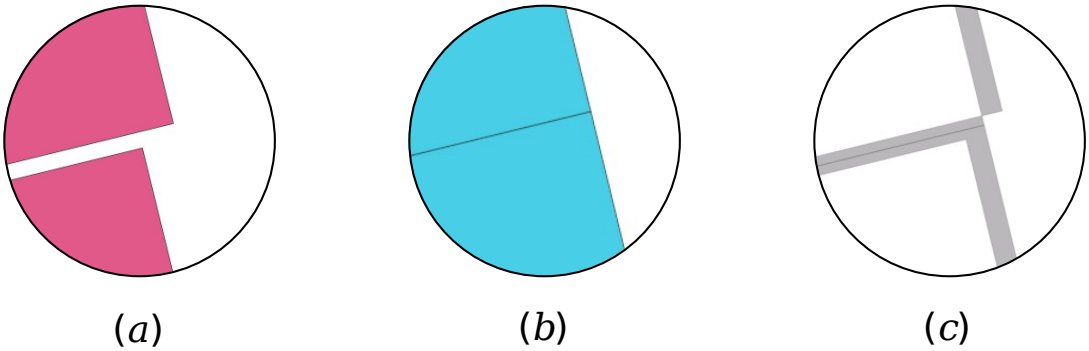


Figure 5.5: An example of how symmetrical difference shows the distortions before and after SR. (a) Input. (b) Rounded result. (c) Symmetrical difference (computed in QGIS)

For the purpose of intuition, simplicity, and ease of implementation, the metric $area\_diff$ is defined as follows:

$$area\_diff = \frac{\sum_{i=1}^{n} \left| \mathcal{A}\left(\mathcal{P}\mathrm{i}\right) - \mathcal{A}\left(\mathcal{P}\mathrm{i}'\right) \right|}{\sum_{i=1}^{n} \mathcal{A}\left(\mathcal{P}\mathrm{i}\right)} \times 100\%. \tag{5.1}$$

where $\mathcal{P}\mathrm{i}$ represents the original input polygon, $\mathcal{P}\mathrm{i}'$ represents the corresponding rounded polygon, $\mathcal{A}\left(\mathcal{P}\mathrm{i}\right)$ stands for the area of $\mathcal{P}\mathrm{i}$ and $\mathcal{A}\left(\mathcal{P}\mathrm{i}'\right)$ denotes the area of $\mathcal{P}\mathrm{i}'$. This metric intuitively quantifies the level of distortions between input and output polygons. It is important to note that there is no strict standard for determining the value at which the distortions can be classified as significant or insignificant. The $area\_diff$ should serve the purpose of making comparisons for one dataset. For instance, when applying different tolerance values with the same set of polygons, the resulting output would vary and the degrees of distortions would differ. The differences can be measured by the value of $area\_diff$. As the value of the $area\_diff$ increases, the corresponding degree of distortion would proportionally escalate. Note however, its significance would be less pronounced when comparing distortions amongst different datasets thus it should only be used for one dataset (with different tolerances).

5.3 Results and Analysis

For the purpose of verifying the correctness and validity of the snap rounded polygons, the datasets are tested with different tolerance values. Corresponding results and statistics are shown to illustrate how polygons are rounded during the SR process and what kind of performance can be expected.

Figure 5.6 depicts an arrangement of two polygons that often occurs in a dataset. At the first glance, from a normal scaling perspective of the two polygons, they would be easily considered as adjacent to each other and only have one common boundary (a line segment). However, the two polygons A and B may not be actually touching if one zooms in enough. In the leftmost figure (Figure 5.6a), there are three light blue rectangles indicating which areas will be enlarged and analysed. Consider the first rectangle, the enlarged details are shown in Figure 5.6b. It can be observed that within the extent vertex o of polygon A is very close to the boundary ac. The distance between them is $0.85cm$ (with the scale of $50:1$), which is usually too small to be a real gap of two "adjacent" buildings (remember that the polygons represent buildings in the real-world dataset), hence this small distance should be eliminated via SR operation. Note that the boundary ac actually consists of two very close line segments, they intersect at vertex a and the distance between them is gradually increasing as they get farther away from a, as is illustrated in Figure 5.6d, the distance between l_1 (belonging to A) and l_2 (belonging to B) is measured as $0.29cm$ (with the same scale $50:1$), which needs to be snap rounded as well. The endpoint of l_2 in Figure 5.6d is vertex e in Figure 5.6f, which is also the endpoint of line segment en (n is shown in the enlarged figure next to Figure 5.6f). One can observe that the details in Figure 5.6f are a bit messy, for the vertex n should be snapped onto the segment l_2 and the gap between segment l_1 and l_2 should be removed. Be aware that snapping n would possibly make vertex e a dangling vertex and leave en a dangling line segment, which is why removing dangling elements process can be necessary sometimes.

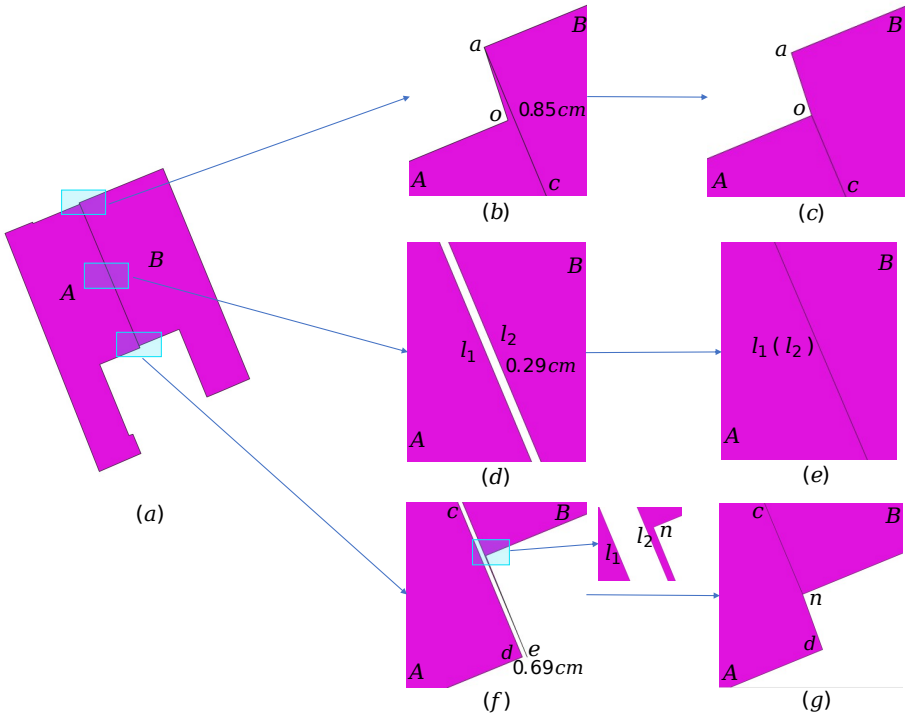


Figure 5.6: An example of resulting polygon arrangement after SR.

The corresponding snap rounded counterparts of the cases described above are illustrated in the rightmost figures of Figure 5.6. Refer to Figure 5.6c, the vertex o and edge ac are snapped together using *snap rounding polygonal vertex and boundaries* schema (explained in the chapter **Methodology**). After rounding procedure, o becomes a vertex that is located on the common boundary between polygon A and B (this also makes o a common vertex). Such behaviour makes the arrangement more robust as vertex o would not unpredictably remove with the existence of rounding errors of floating-point arithmetic (e.g. for the original position shown in Figure 5.6b, o can shift to the other side of ac, resulting in polygon overlaps). As for the small gap between segment l_1 and l_2, it will be removed after the rounding, making l_1 and l_2 become one segment owned by both A and B. Regarding the case of Figure 5.6f, vertex n and segment l_2 are firstly snapped together, afterwards the (dangling) vertex e is removed and the small gap between l_1 and l_2 gets eradicated. The resulting arrangement is cleaner and more stable as is shown in Figure 5.6g. It should be noted that vertex n also becomes a common vertex of polygon A and B after SR.

Figure 5.7 shows another exemplary instance: a polygon containing an inner hole which is very close to the exterior of the polygon. As is illustrated in Figure 5.7a, the input polygon A has an inner hole τ which is very close to the polygon boundary e_1e_2. See Figure 5.7b, m and n are two vertices of τ and mn is denoted as one of the hole's boundaries. Let us denote the projection of mn on the polygon boundary e_1e_2 as l_1. In accordance with the experimental results, the average distance between segment mn and segment l_1 is nearly $3.58cm$. Similar to Figure 5.6d, such distance is also too small to be a real gap between a hole and the exterior of a polygon thus it requires rounding operation as well. Through the utilization of *snap rounding polygonal vertex and boundaries* schema, vertex m, n and segment

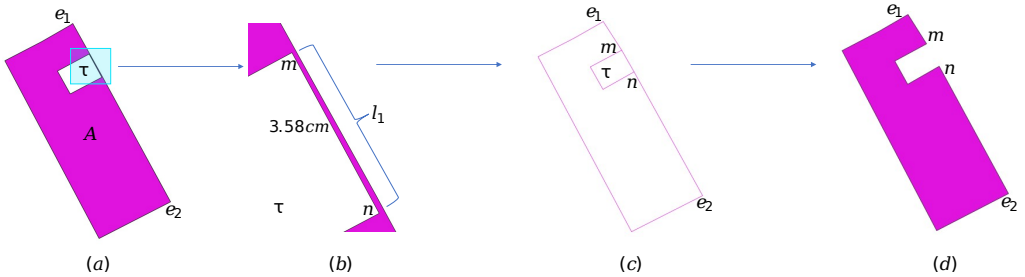
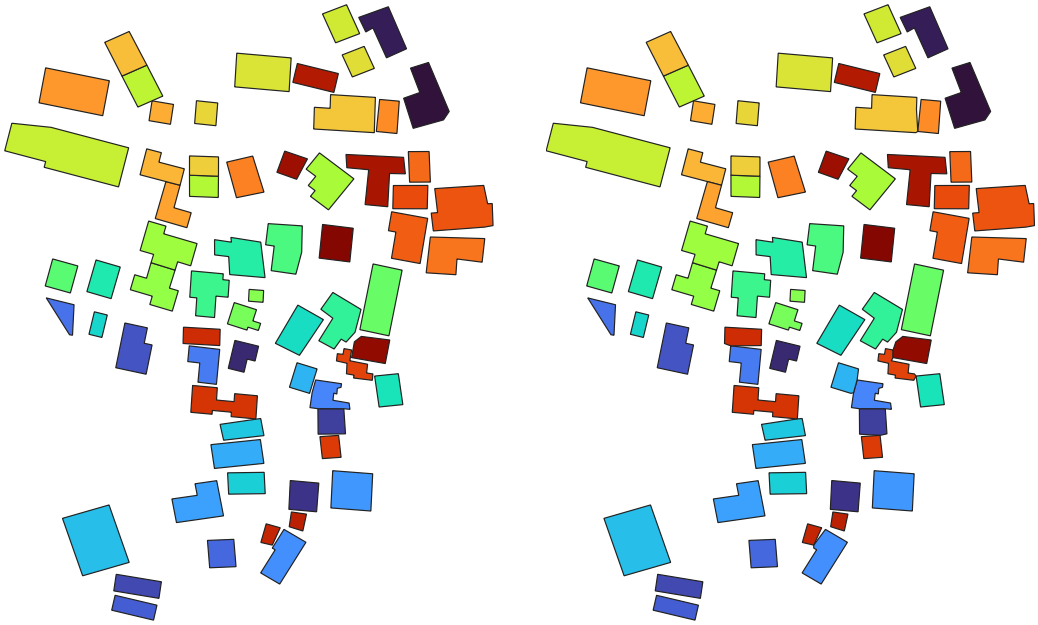


Figure 5.7: An example of snap rounding a polygon with a hole. (a) The input polygon, with a inner hole denoted as τ. (b) Enlarged area of the margin of the inner hole τ. Segment mn is one of the boundaries of τ and is very close to the polygon exterior. l_1 is the projection of mn on the exterior (the length of l_1 is equal to the length of mn and l_1 is part of the exterior). (c) The snap rounded boundaries of the polygon, with mn being both the exterior of the polygon and the boundary of the inner hole, which means the hole and the polygon are touching. (d) Reconstructed polygon (resulting polygon) from the boundaries depicted in (c).

l_1 are snapped together, resulting the rounded shape depicted in Figure 5.7c. Because of the rounding, segment mn replaces l_1 and becomes part of the exterior of the polygon, meanwhile mn is still the boundary of τ. A polygon with such shape is actually an invalid polygon, see [Ledoux et al., 2012] for details. The focus of this thesis is not handling the invalidities thus further discussion will not be presented here. In the reconstruction stage, the *polygonizer* (mentioned in the chapter **Methodology**) automatically handles the possible invalidities and yield valid polygons as output most of the time. The resulting polygon shown in Figure 5.7d is valid and more robust, with the changes of original topology as a price.

Figure 5.8 shows an example of a set of polygons. The polygons are part of the Faroe Islands dataset (downloaded from GeoFarbik [GeoFabrik, 2023] and projected using `EPSG:3575 - WGS 84 / North Pole LAEA Europe`). There exists multiple cases that require SR operation in the original dataset (Figure 5.8a). The rounded result (with the tolerance $0.5m$) is shown in Figure 5.8b. The polygons are coloured in conformity with the corresponding ID information so as to intuitively show whether the ID information are kept accordingly. For instance, polygons with the same color in Figure 5.8a and Figure 5.8b represent the same polygon before and after SR (with the same ID). In accordance with the observation, the distortions before and after SR are negligible (with $area\_diff = 0.002$) and the ID information of the polygons has been correctly preserved. In order to demonstrate the differences between the input and the result more specifically, some cases are enlarged and displayed with more details. See Figure 5.9, case $(a), (b), (c), (d), (e), (f)$ illustrate the close elements under a certain tolerance. It is of importance to know that some cases include both *close polygonal vertices* and *close polygonal vertex and boundaries*. Take Figure 5.9a for an example, the bottom vertex of the orange polygon is both close to the bottom vertex of the yellow polygon and the boundary of the yellow polygon. As previously described, SR will start to proceed from the minimum case hence the vertex (of the orange polygon) and the boundary (of the yellow polygon) will firstly be snapped together, the snapping result is depicted in Figure 5.10a. In Figure 5.9e there only exists *close polygonal vertex and boundaries* case as the distance from the vertex of the red polygon to any vertex of the blue polygon is larger than the given tolerance. The snapped counterpart is shown in the right figure of Figure 5.10e. The rounding results



(a) (b)

Figure 5.8: An exemplary dataset: an excerpt of Faroe Islands. The polygons are coloured according to the ID information. (a) Input polygons. (b) Polygons after SR.

of other cases are also shown in Figure 5.10.

5.4 Benchmarking

In order to assess the overall performance of the developed prototype, different tolerance values have been tested. The basic statistics are shown in Table 5.2. The prototype runs robustly for different datasets. The overall run time varies due to multiple reasons. Generally speaking, the run time mainly depends on the number of input polygons, the total runtime increases as the number of polygon increases. As observed in Figure 5.11, the total run time varies in accordance with the size of the dataset (i.e. the number of polygons) with the same tolerance. To illustrate this point, consider a tolerance value of $0.01m$. When the datasets are arranged in ascending order based on the run time, the sequence is as follows: DHC - Faroes - Delft - Freiburg - RCD. This order corresponds to the number of polygons in each dataset, with DHC (19878), Faroes (30926), Delft (38376), Freiburg (53723), and RCD (127795). This holds true for all tested tolerances (ranging from $0.00m$ to $0.09m$). Figure 5.11 solely focuses on illustrating the differences regarding the run time for different datasets. It should be pointed out that different datasets require different amounts of time to a large extent. For instance, the smallest DHC dataset only requires approximately 3-5 minutes, while the largest RCD dataset demands close to 195-200 minutes. As a result, the specific changes in runtime for each dataset displayed in the figure appear relatively uniform. However, this is not the case when the run time is displayed with regard to each individual dataset, as illustrated in Figure 5.12.

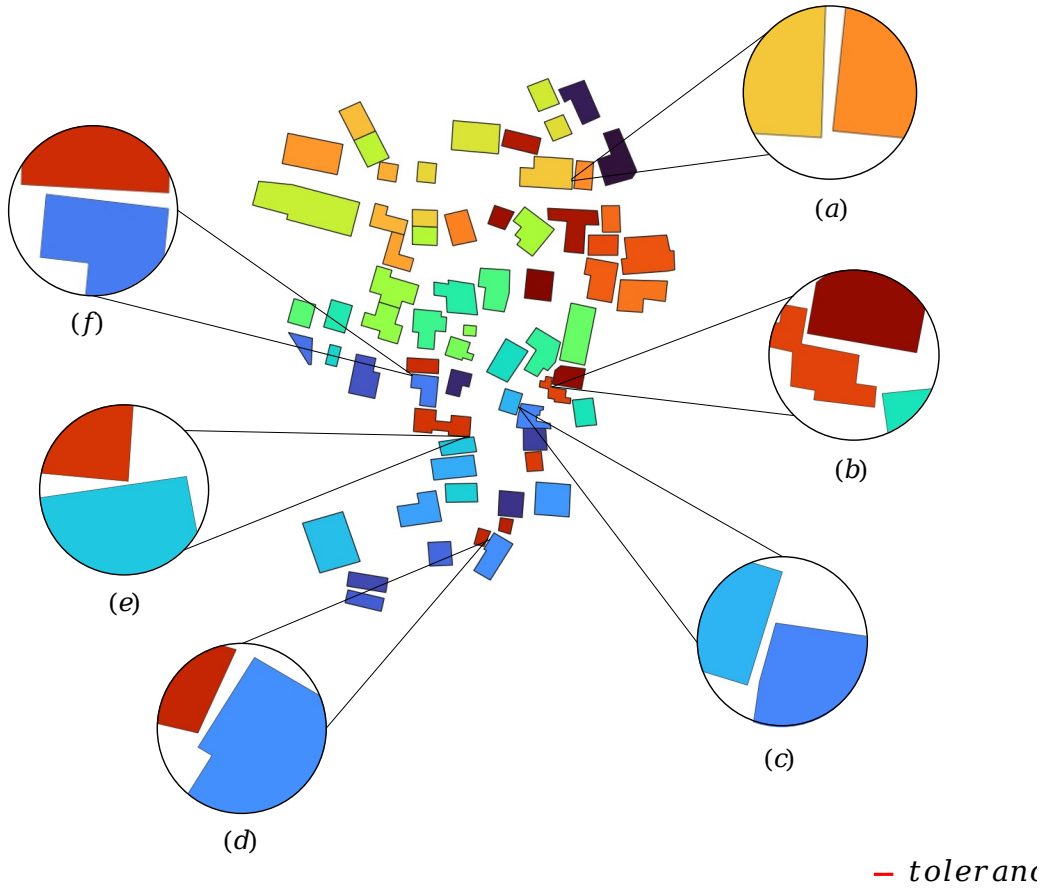


Figure 5.9: Some SR cases in Faroe Islands dataset.

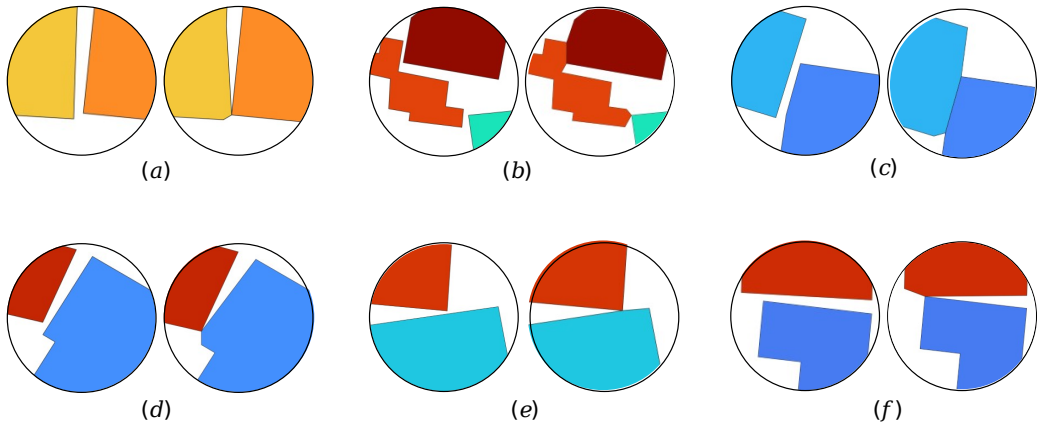


Figure 5.10: Before and after SR operation of the cases shown in Figure 5.9. In each sub figure, the left one shows the original input and the right one shows the rounded result.

Dataset	Number of polygons	Tolerance(m)	Total run time(min)
DHC	19878	0.00	3.267
DHC	19878	0.01	3.908
DHC	19878	0.02	4.096
DHC	19878	0.03	4.242
DHC	19878	0.04	4.382
DHC	19878	0.05	4.566
DHC	19878	0.06	4.905
DHC	19878	0.07	5.173
DHC	19878	0.08	5.212
DHC	19878	0.09	5.419
Dataset	**Number of polygons**	**Tolerance(m)**	**Total run time(min)**
Faroes	30926	0.00	6.159
Faroes	30926	0.01	7.088
Faroes	30926	0.02	7.263
Faroes	30926	0.03	7.354
Faroes	30926	0.04	7.611
Faroes	30926	0.05	7.725
Faroes	30926	0.06	8.056
Faroes	30926	0.07	8.245
Faroes	30926	0.08	8.543
Faroes	30926	0.09	9.514
Dataset	**Number of polygons**	**Tolerance(m)**	**Total run time(min)**
Delft	38376	0.00	17.090
Delft	38376	0.01	19.130
Delft	38376	0.02	19.223
Delft	38376	0.03	20.558
Delft	38376	0.04	22.050
Delft	38376	0.05	25.145
Delft	38376	0.06	27.686
Delft	38376	0.07	30.051
Delft	38376	0.08	31.593
Delft	38376	0.09	33.650
Dataset	**Number of polygons**	**Tolerance(m)**	**Total run time(min)**
Freiburg	53723	0.00	32.305
Freiburg	53723	0.01	34.153
Freiburg	53723	0.02	35.036
Freiburg	53723	0.03	35.283
Freiburg	53723	0.04	36.072
Freiburg	53723	0.05	36.253
Freiburg	53723	0.06	36.631
Freiburg	53723	0.07	37.018
Freiburg	53723	0.08	37.232
Freiburg	53723	0.09	37.368
Dataset	**Number of polygons**	**Tolerance(m)**	**Total run time(min)**
RCD	127795	0.00	194.148
RCD	127795	0.01	196.729
RCD	127795	0.02	196.970
RCD	127795	0.03	197.022
RCD	127795	0.04	197.056
RCD	127795	0.05	197.583
RCD	127795	0.06	197.960
RCD	127795	0.07	198.275
RCD	127795	0.08	198.323
RCD	127795	0.09	198.875

Table 5.2: Run time of selected datasets regarding different tolerance values.

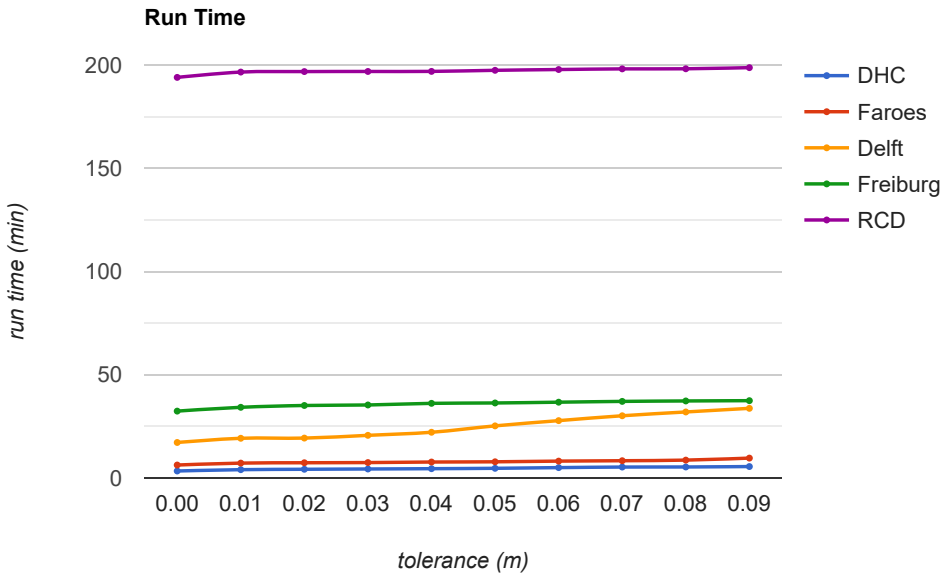
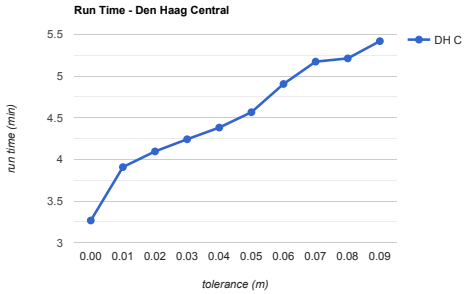


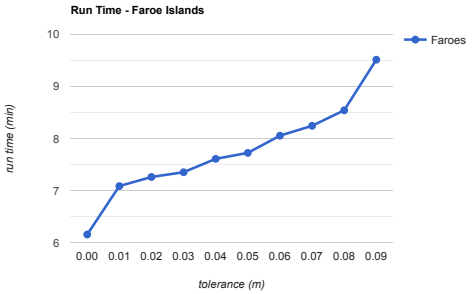
Figure 5.11: Running time of selected datasets with different tolerances.

Figure 5.12 illustrates the influence of tolerance on the overall runtime of the tested datasets. Typically, as the tolerance value increases, the runtime also tends to increase. This can be attributed to the fact that larger tolerance values usually introduce a greater number of SR cases, thus consuming more time. Furthermore, it is important to mention that there is no explicit pattern between the run time and the tolerance values, it is neither linear nor regular. It is also worth noting that the run time is not equal to 0 even if the tolerance is set to 0.00. This is because other procedures (such as reading the input, tagging, output) would also require a certain amount of time.

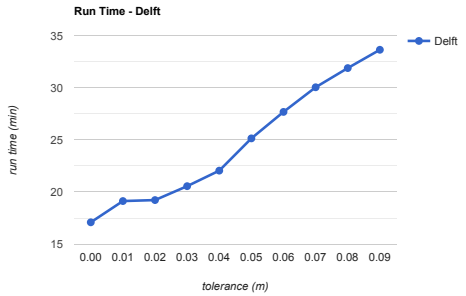
As outlined in the methodology, the SR process primarily consists of two key steps: tagging the entire triangulation and snapping polygonal elements, including vertices and boundaries. To better evaluate the efficiency and gain a better understanding of the proposed approach, run time measurements are conducted and categorized accordingly. Throughout the experiments, a discovery has been found that tagging process would consume the majority of the total run time hence the resulting figure is made according to the time used for tagging and the total run time (allowing for more straightforward observation). The statistical findings are presented in Figure 5.13. Figure 5.13a demonstrates the time allocation for tagging (represented in red) and the overall runtime (depicted in blue) for each dataset. Figure 5.13b shows the ratio of time consumed by tagging in relation to the total runtime. These results indicate that the tagging stage typically accounts for a minimum of 80% or more of the total run time. The average tagging ratio of the selected datasets, occupies approximately 89.9% of the total time. Consequently, it can be concluded that that the tagging stage is the most time-consuming aspect regarding the SR process. This observation primarily stems from the utilization of the BFS algorithm on all input polygons. BFS itself is an complex operation, and its extensive use in the tagging process renders it more



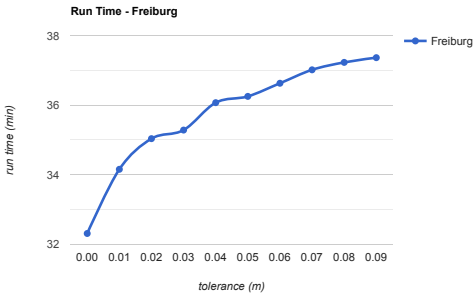
(a) DHC (19878 polygons)



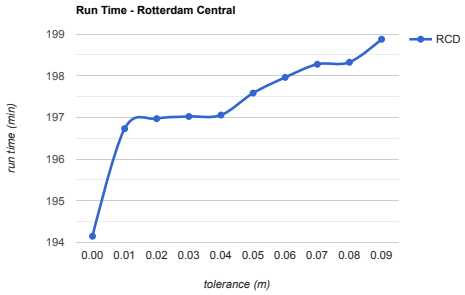
(b) Faroes (30926 polygons)



(c) Delft (38376 polygons)

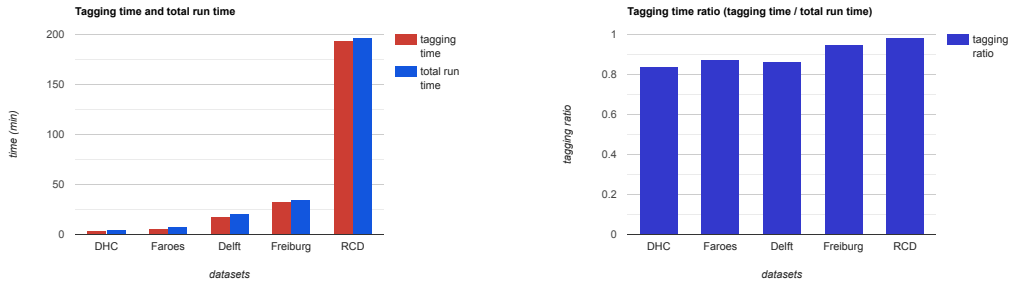


(d) Freiburg (53723 polygons)



(e) RCD (127795 polygons)

Figure 5.12: Run time regarding different tolerance values.



(a) (b)

Figure 5.13: Tagging time and total run time. (a) Tagging time and total runtime regarding the selected datasets ($tolerance = 0.01m$). (b) Tagging time ratio (tagging time / total run time) regarding the selected datasets ($tolerance = 0.01m$).

computationally expensive.

To quantitatively assess the distortions introduced by SR in the datasets, area difference was measured using different tolerance values. The results of these tests are presented Figure 5.14 and Figure 5.15. The first figure shows the measured area differences of five testing datasets. It can be observed that, for a given tolerance, the area difference generally correlates with the size of the datasets, as demonstrated in Figure 5.11. For instance, when a tolerance of $0.09m$ is used, the dataset with the fewest polygons (DHC) exhibits an area difference of approximately 0.1%, while the largest dataset (RCD) shows an area difference of nearly 3.0%. As previously mentioned, larger area differences correspond to more significant distortions. Among the five datasets, the one with the largest number of polygons normally contains a greater number of SR cases within a certain tolerance. SR process involves necessary modifications of the original shape of polygons to eliminate small gaps, hence distortions are inevitably introduced. A special case is when the tolerance is set to zero, resulting in no modifications being made to the polygons from the input dataset, thus yielding an area difference of zero as well. However, as soon as modifications are introduced, the area difference begins to increase rapidly, as depicted in Figure 5.15, from tolerance 0.00 to tolerance 0.01. Regarding the trend of increase, it varies among the different datasets. In the case of Delft, Freiburg, and RCD datasets, the area difference exhibits a relatively slow increasing as the tolerance value increases. This suggests that even with a larger number of cases processed, the distortions of the polygons are relatively small. Conversely, the trend observed in the DHC and Faroes datasets (refer to Figure 5.15a and Figure 5.15b) demonstrates a steeper increase, indicating that as the tolerance value increases, the degree of distortions becomes more significant.

In order to determine the scalability of the prototype across different sizes of datasets, various tests have been conducted based on the selected datasets. Primarily, the number of polygons has been manually adjusted within a range of 10 to more than $100,000$. Furthermore, additional statistics were documented as well, including the number of vertices, segments and faces within the triangulation. The testing results are shown in Table 5.3. The statistics clearly demonstrate a direct correlation between the number of polygons (or elements) and the runtime, as the runtime consistently increases with a higher number of polygons (or elements). These statistics serve as the basis for generating informative line

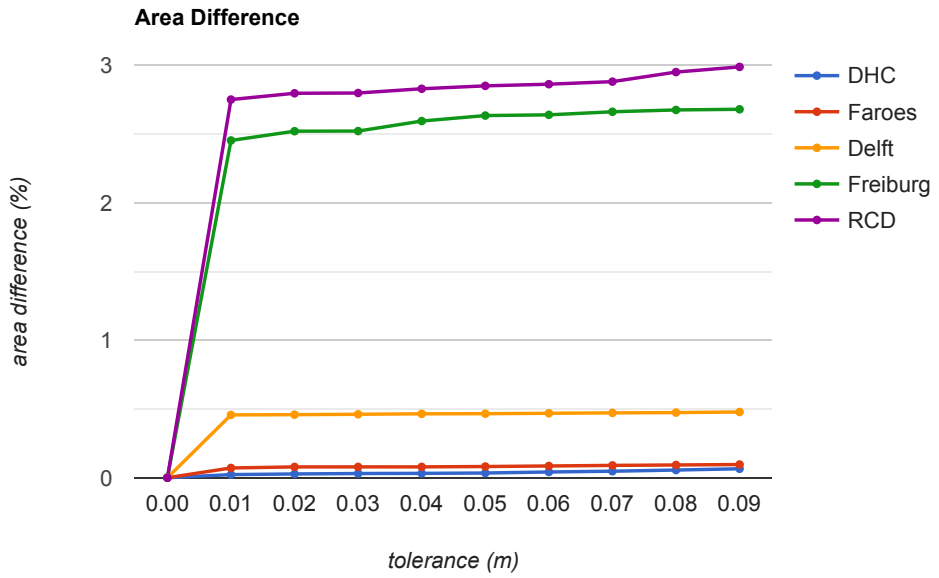


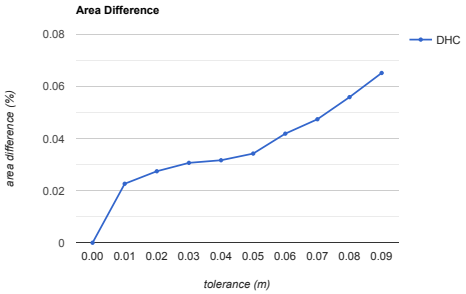
Figure 5.14: Area difference of selected datasets with different tolerances.

charts that effectively illustrate the trends in runtime over the datasets. It is important to understand that the number of polygons may not be the primary factor that significantly affects the runtime. Polygons usually have different types of shapes, which can result in different numbers of elements (vertices and segments). For instance, observed in the DHC and Faroes datasets in Table 5.3, even when the polygon counts are the same, the number of vertices and line segments can differ due to differences in polygon shapes.

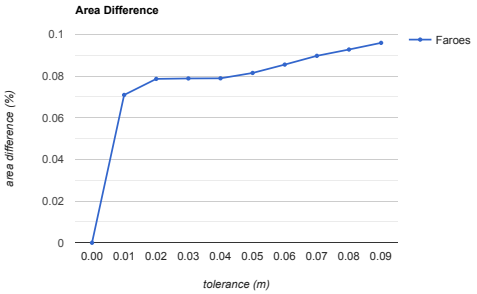
Figure 5.16 depicts the relationship between the total runtime and the dataset sizes (represented by the number of polygons). The five figures in the graph collectively demonstrate a consistent trend: as the number of polygons increases, the runtime also increases. Moreover, these figures also illustrate the scalability of the developed prototype, showing it is capable of handling datasets of varying sizes.

The number of polygons serves as a rough indication of the size of datasets. However, in the common practice, considering the specific shapes (of polygons), it provides only a general overview. In the methodology, it has been mentioned that all the input polygons need to be embedded into a triangulation first and in the tagging stage all the faces in the triangulation are traversed to tag the triangulation. Therefore, using the number of faces may be a more appropriate way to represent the actual size and complexity level of the datasets. This choice is also motivated by previous findings, which revealed that the tagging stage dominates the time required for the SR operation (as shown in Figure 5.13). Given the fact that the shape of all the faces are guaranteed to be triangles, the number of faces provides a more accurate measure of the dataset's scale. Two examples are presented to illustrate the differences in the number of faces despite having the same number of polygons.

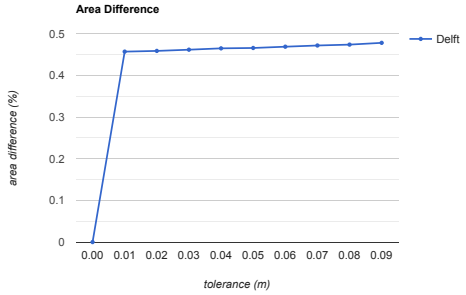
To have a clear understanding of the differences among the selected datasets having the



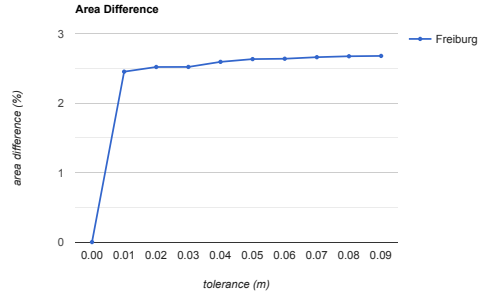
(a) DHC (19878 polygons)



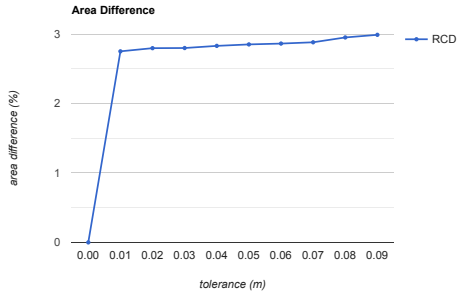
(b) Faroes (30926 polygons)



(c) Delft (38376 polygons)



(d) Freiburg (53723 polygons)

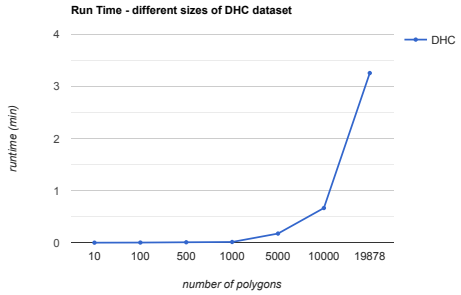


(e) RCD (127795 polygons)

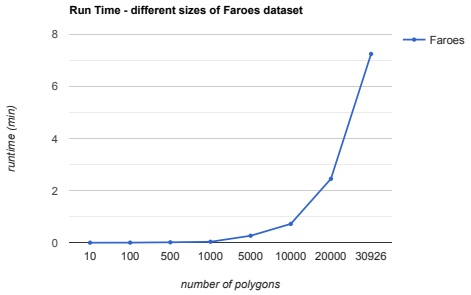
Figure 5.15: Area difference regarding different tolerance values.

Num of polygons (DHC)	Num of vertices	Num of segments	Num of faces	Runtime (min)
10	383	1125	743	0.00168242
100	1715	5118	3404	0.00380131
500	6545	19603	13059	0.00956036
1000	8491	25441	16951	0.0147298
5000	29981	89911	59931	0.176387
10000	57252	171718	114467	0.666789
19878 (total)	115589	346735	231147	3.25784
Num of polygons (Faroes)	**Num of vertices**	**Num of segments**	**Num of faces**	**Runtime (min)**
10	79	224	146	0.00257392
100	744	2214	1471	0.00498292
500	3677	11012	7336	0.018107
1000	7425	22252	14828	0.0376588
5000	30913	92721	61809	0.268987
10000	58106	174301	116196	0.724415
20000	115388	346146	230759	2.45576
30926 (total)	174557	523642	349086	7.24895
Num of polygons (Delft)	**Num of vertices**	**Num of segments**	**Num of faces**	**Runtime (min)**
10	148	432	285	0.00254325
100	2872	8600	5729	0.0066124
500	11852	35525	23674	0.0382818
1000	18810	56396	37587	0.0857326
5000	33377	100092	66716	0.342467
10000	69117	207312	138196	1.13968
20000	122748	368210	245463	3.91653
30000	173797	521356	347560	10.9141
38376 (total)	210917	632723	421807	18.0034
Num of polygons (Freiburg)	**Num of vertices**	**Num of segments**	**Num of faces**	**Runtime (min)**
10	391	1154	764	0.00361835
100	1854	5539	3686	0.00564597
500	6008	18009	12002	0.0220264
1000	10131	30376	20246	0.0453322
5000	39743	119203	79461	0.354143
10000	78735	236176	157442	1.18357
20000	148623	445838	297216	4.89471
30000	201209	603598	402390	11.7602
40000	243381	730114	486734	19.7922
53723 (total)	300126	900347	600222	35.5441
Num of polygons (RCD)	**Num of vertices**	**Num of segments**	**Num of faces**	**Runtime (min)**
10	337	987	651	0.00259865
100	1653	4936	3284	0.00538976
500	10019	30031	20013	0.0324458
1000	16146	48412	32267	0.0726439
5000	39212	117610	78399	0.423932
10000	75164	225464	150301	1.38921
50000	246332	738964	492633	25.2106
100000	482126	1446343	964218	123.575
127795 (total)	616293	1848840	1232548	204.754

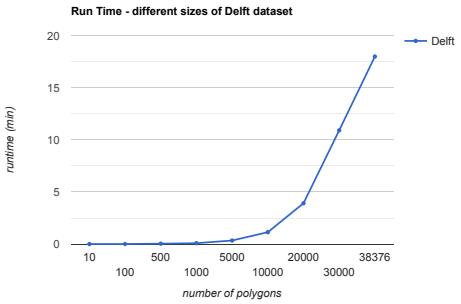
Table 5.3: Run time of different sizes of selected datasets with tolerance $0.01m$



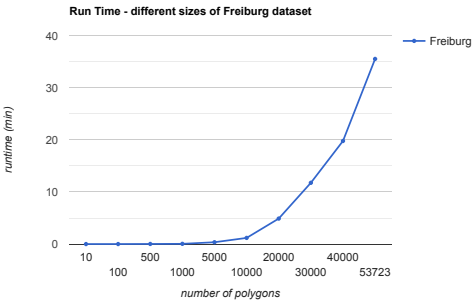
(a) DHC (19878 polygons)



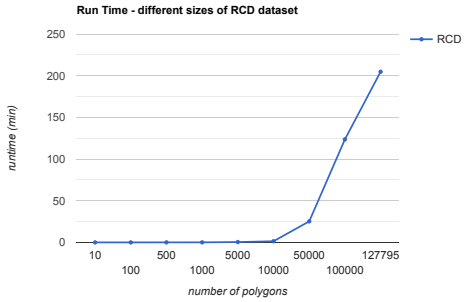
(b) Faroes (30926 polygons)



(c) Delft (38376 polygons)

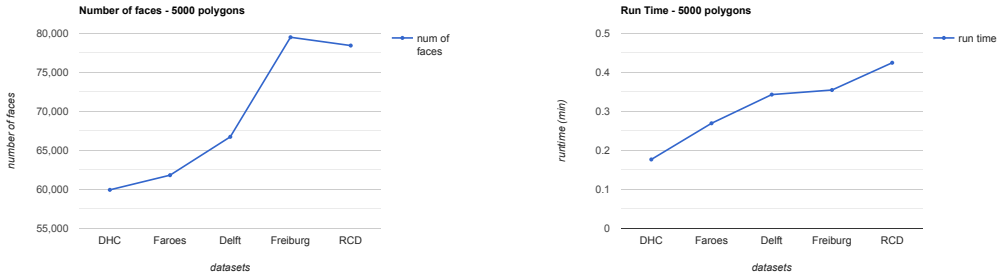


(d) Freiburg (53723 polygons)



(e) RCD (127795 polygons)

Figure 5.16: Run time regarding different sizes of datasets (tolerance = 0.01m).



(a) (b)

Figure 5.17: Datasets having the same number of polygons. (a) The number of faces in the embedded triangulation regarding different datasets. (b) The total run time regarding different datasets.

same number of polygons, the number of polygons is set to 5000. It is worth mentioning that if the number is too small, the differences may be insignificant, and if it is too large (e.g. 100,000), not every dataset would contain such a large number of polygons. As depicted in Figure 5.17a, all datasets consist of 5000 polygons, yet the number of faces in the triangulation varies significantly, ranging from approximately 60,000 to nearly 80,000, which revels substantial differences among the different datasets. This indicates that datasets with a higher number of faces tend to possess more vertices and line segments, implying that the polygons have more complex (and possibly irregular) shapes. Figure 5.17b illustrates the run time for selected datasets (each dataset contains 5000 polygons). It appears that the run time increases as the number of faces increases for the DHC, Faroes, Delft, and Freiburg datasets. However, despite having fewer faces than the Freiburg dataset, the RCD dataset shows a significant increase in run time. This can be attributed to the fact that the run time is not solely determined by the dataset size (as indicated by the number of faces in this case). The total number of SR cases within a specific tolerance also impacts the run time. Although the snapping process only consumes a small portion of the overall runtime (approximately 20% as the tagging stage consumes more than 80% based on previous testing results), if the number of SR cases is substantial, it will also inevitably consume a certain amount of time. Hence, although the number of faces in RCD dataset is less than that in the Freiburg dataset, the run time still increases.

In addition to the number of polygons and faces in the triangulation, the number of vertices and segments can also serve as indicators to gain an understanding of the scale and complexity level of the dataset. However, it is important to note that relying on the number of vertices or line segments may provide an incomplete measurement. It is also challenging to create a meaningful index by combining these two factors reasonably. On the other hand, although the number of polygons may not be the most precise metric, it remains the most apparent and straightforward indicator for users, especially when there are significant differences in numbers (e.g. 100 and 10,000).

6 Conclusion and future work

6.1 Conclusion

This thesis firstly reviewed the main existing snap rounding algorithms related to a set of line segments and studied the use cases of applying a *constrained triangulation* to fix geometries. Subsequently a novel method aiming at rounding 2-dimensional polygons was proposed, implemented and tested. The developed approach, *snap rounding on polygons*, processes input polygon arrangement in a fully automatic manner thus requires no human interference.

The proposed method commences with embedding the input polygons into a *constrained Delaunay triangulation*. Afterwards the triangulation is tagged and a list of constraints is established to represent the boundaries of polygons. The snap rounding process is divided into two main categories: *snap rounding polygonal vertices* and *snap rounding polygonal vertex and boundaries*. These two cases are gradually handled from the *minimum case*, which has the smallest distance between elements (including vertices and line segments) under the certain tolerance. During the modification procedure the list of constraints and the triangulation are updated dynamically to keep track of the information of polygons. In other words, the boundaries of polygons are maintained in an adaptive manner hence all polygons can be reconstructed and output at any step. Last but not least, the *polygonizer* of `GEOS` library is utilized to reconstruct polygons from the list of constraints. In the rounded arrangement, both the distance between vertices and the distance between vertex and boundaries are guaranteed to be greater than a certain value (i.e. the predefined tolerance). Therefore a set of more robust and cleaner geometric objects can be expected in the final result.

Up to this point, the primary research questions can be answered as follows:

- **How to integrate a CT with the input polygons (possibly with interior holes)?**
  Input polygons can be embedded into a triangulation (i.e. a CT) to realise the integration. Since the boundaries of polygons should not be unintentionally changed(that being said, boundaries of polygons should not be modified by the CGAL triangulation package and should be only modified and updated manually), the boundaries representing the polygon shapes must be guaranteed not to change automatically. Therefore they are embedded as constrained edges in a CT. In the implementation, a CDT is actually used for the purpose of making the triangulation as good-shaped as possible to avoid possible skinny triangle shapes while performing the SR process. As mentioned earlier, in a CDT constrained edges will not change even if their presence make the triangulation not satisfy the Delauny criteria. This trade-off is necessary to uphold the integrity of the polygon shapes while maintaining a well-shaped triangulation. Another benefit is that embedding polygons via their boundaries can handle intricate shapes, such as a polygon with multiple interior holes, and possible invalidities, such as a polygon with self-intersections. There will be no issues when inserting the boundaries as constrained edges hence this approach is robust for various input.

- **Is it better to have a certain type of a CT such as a CDT or is any type of a CT fine?**
  Principally any kind of a CT would suffice. However a CDT is preferred as it is capable to automatically handle the possibly irregular shapes that may arise during the SR process and avoid the skinny triangles as much as possible. It is important to know that SR would alter the triangulation by modifying the constrained edges. This procedure may generate undesirable triangles, including the aforementioned skinny shapes. The algorithm needs to check whether a skinny triangle is a liver triangle (recall that a sliver triangle means a vertex is possibly very close to a boundary and such case may necessitate the application of the SR process. Therefore, minimizing the occurrence of unintended triangles (such as skinny triangles) would be helpful to enhance the efficiency of the prototype.

- **How to link the triangulation with the original polygons? If the triangulation is modified, how to update the changes to the polygons?**
  Use a separate container (e.g. a list) to store the boundaries of polygons (along with the related information). This allows for non-interfering and dynamic updates to both the boundaries and the triangulation. More specifically, during the SR process, cases that require SR operations are identified within the triangulation, then the polygonal boundaries in the list are modified accordingly. Afterwards the structure of the triangulation is also updated to reflect these changes. By adopting this approach, modifications to the polygons and the triangulation can be made dynamically and synchronously.

- **Polygons usually have attributes attached (e.g. polygon id, area of a polygon), how to preserve them in a proper way during the SR process?**
  For each polygon's boundary, a constraint is constructed and stored in a list (as previously mentioned, a list is utilized to store the polygonal boundaries). Meanwhile, a tag, typically the polygon's ID, is attached to each constraint to indicate which polygon it should belong to. This ensures that after modifying the boundaries and the triangulation, it is still possible to determine which constraints represent which polygons. The tag also serves as a key, allowing for querying of other relevant information. For instance, assume in the input dataset each polygon has a code and a type. These information are stored in a table based on the unique ID of each polygon. The modifications are limited to the geometric aspects of the boundaries, specifically the coordinates of the endpoints. The attached ID tags are preserved without any modifications. In the reconstruction stage, the preserved ID information can be utilized to reconstruct the polygons and retrieve associated attributes.

- **Given a certain threshold, there may exist several SR cases (e.g. multiple polygonal vertices and polygonal vertex and boundaries), what is the most reasonable order when SR is being performed?**
  In scenarios involving multiple SR cases, the order in which the processing sequences are implemented can have a significant impact on the final results and potentially lead to cascading effects. To address this issue, the SR process is initiated with the case that exhibits the smallest gap within a certain tolerance. In other words, the distance between snapping elements, such as a vertex and a boundary or between two vertices, is the smallest one in comparison to the gaps in all other SR cases. Such approach helps to prevent cascading effects that may ar ise during SR. For instance, if SR starts with a case that has a large distance, it can potentially result in the creation of newly shaped constraints that intersect with existing ones, as illustrated in the Methodology chapter. Therefore, based on the testing and analysis, commencing the SR process from the case with the minimum gap is deemed as the optimal approach. This strategy

minimizes the likelihood of creating unwanted cascading effects (such as intersections) and ensures a more reliable outcome.

- **How to measure and evaluate the distortions of the polygons before and after SR?** To gain a more comprehensive understanding of the potential distortions resulting from SR, two approaches (symmetrical difference and area difference) are introduced with the aim to provide valuable insights. Symmetrical difference in QGIS would allow for easy identification and visualization of the distortion areas and help to have an better understanding of what has been changed before and after SR. To quantitatively measure the magnitude of distortions, area difference is introduced. This metric provides an overall numerical assessment of the differences between the input dataset and its rounded counterpart. It has been found that the area difference increases as the tolerance value increases, which indicates that larger tolerances result in greater distortions. This is rather reasonable because large tolerance values will lead to more SR cases, hence result in more polygons that need to be modified. Consequently, the higher the number of modified polygons, the more pronounced the distortions become.

The testing results imply that the small gaps between vertices or between vertex and boundaries are removed by the rounding operation. The implemented prototype is capable of handling not only valid geometries but also invalid ones, such as overlapping area. The originally topological and geometric characteristics are preserved as much as possible to minimize the distortions caused by the snap rounding. It should however be noted that the distortions are natural and can not be completely avoided regarding the rounding process. Different evaluation methods has been proposed to measure the quality of the result, including symmetrical difference and area difference. Generally speaking, the quality of the result is deemed to be higher as the magnitude of distortion is smaller. The developed prototype possesses the ability to work with datasets of different size, e.g. from 100 polygons to $100,000$ polygons. Theoretically the prototype can also handle large dataset such as more than $1,000,000$ polygons. Notwithstanding, it should be warned that processing large dataset would require more computational resources and would be computational-intensive.

The developed prototype has also been tested multiple times to assess its reliability and capability to handle datasets of varying scales. Aiming at this five different kinds of datasets were selected for testing purposes. Based on the results, it was observed that the most time-consuming process was the tagging of the triangulation. In fact, across all five datasets, the tagging stage accounted for over 80% of the total run time. This could be optimized by two different kinds of strategies. The first approach is to use a more efficient container to store all tagged constraints and improve the efficiency of checking redundancies during the tagging process (recall that when each constraint is being added, it has to be checked whether it has already existed in the list to avoid duplicates). Optimizing this would significantly improve the overall efficiency of the tagging stage. Another approach requires structural changes of the overall work flow. Remind that the purpose of the tagging process is to obtain a set of constraints (with polygon ID information attached) to represent the boundaries of the polygons. This step can also be realised when loading the polygons from datasets into memory. The boundaries of the polygons can be read and stored with the correct ID information, which eliminates the need for utilizing the BFS algorithm and noticeably reduces the overall runtime. However, adopting this process may limit the potential for future expansion of the prototype. For instance, handling the overlap areas can be further expanded based on the current tagging strategy. The overlap area can be identified by assigning multiple tags to a face in the triangulation (recall that in the tagging stage each

face is checked). Extracting the boundaries directly while reading the file would not allow for handling the overlap areas and is not versatile enough. Therefore even if using BFS may not be the fastest solution, it can allow for more possibilities in the future development.

6.2 Future work

Albeit the proposed method is implemented and has a certain degree of robustness, there are still some areas that can be improved or expanded to achieve a wider application range and higher efficiency. The following aspects are regarded as possibly prospective directions for future work:

- **Support for rounding polygons and line segments**.
  In current implementation, the output of SR is polygon. However, it is observed that during the SR process some polygons may collapse into line segments or even points. For instance, a very skinny polygon can become a simple line segment (retaining the associated attributes) after SR process. Although such geometry collapse is permitted within the process, the collapsed geometries are not yet written to the output file. Future enhancements aim to support the representation of collapsing polygons as line segments in the output file. Additionally, it is also desirable to extend the support for snapping both polygons and line segments together. This is particularly relevant as input datasets may consist not only of polygons but also include line segments, such as datasets containing buildings, roads, and rivers. It can be possible that a building is constructed next to a road or a river, meaning the road or the river should be touching the building. There may also exhibit small gaps between roads (or rivers) and buildings that require SR operation to remove the gaps.
- **More flexible processing of overlapping areas**.
  In the tagging stage, faces can also be tagged. This would allow for adding more support of handling overlapping area. For instance, if a face is tagged more than once, then this face is known to be part of the overlapping area, based on which further operations can be implemented.
- **Improve efficiency by utilizing `std::unordered_set` instead of `std::list` to store the constraints**.
  In current implementation a `std::list` is utilized to store all the boundaries (constraints). Albeit `std::list` can provide relatively fast insertions and deletions, it is not very efficient when one wants to find a certain element in the list. To be more precise, `std::list` is internally realised via a linked list, hence it does not provide direct random access to its elements (note that direct random access is usually provided by array-like containers like `std::vector`). When finding a certain element, a linear traversal of the list is necessary. The finding process iterates through each element in the list from the beginning until the desired element is found or the end of the list is reached. This will result in a time complexity of $O(n)$, where n is the size of the list (i.e. the number of the stored elements).

  In C++, there exists a data structure `std::unordered_set` which has the ability to provide constant time complexity (average-case) for operations like searching, insertion, and deletion. It is internally implemented using a hash table hence permits for fast querying compared to `std::list`. The time complexity of finding elements in a `std::unordered_set` is typically $O(1)$ on average. Note however, this is based on the

fact that he hash function is well-distributed and there are no significant collisions. If the number of collisions is high, the time complexity can become $O(n)$.

It is important to note that naturally `std::unordered_set` only supports for built-in data types. Storing customized objects (e.g. constraints) would require constructing customized hash functions.

- **Improve efficiency by using additional data structures**.
  Each time a case is processed, the triangulation would automatically update. Although the changes are ensured to be local by the `CGAL triangulations` package, new snapping cases can be created due to the cascading effects and in the next loop a traversal is still required to find the next minimum case. This issue may be solved by utilizing data structure such as `std::priority_queue` to store the snapping cases in accordance with the distance.

- **Improve robustness for large tolerance values.**
  SR usually works for appropriate tolerance values. However in practice, large tolerance value can be given and it may cause undefined behaviours of the prototype. The mechanism of SR is designed for handling closely positioned vertices and boundaries. If the tolerance values is too large, the geometry of polygons would collapse and cause crashing of the prototype. During the experiments, the use of large tolerance values results in the prototype entering an infinite loop and it then has to be shut down manually. Therefore, enhancing the robustness of the prototype to handle large tolerance values is an important area for future improvement.

- **More automated process**
  The downloaded dataset can not be directly used in the prototype as the coordinates are presented using longitude and latitude. It needs to be re-projected with a suitable CRS to convert the coordinates into XY form. This process can be integrated into the prototype if the CRS can be known in advance. Additionally, integrating auto-correction techniques can further enhance the prototype's automation capabilities, such as integrating *prepair* (a developed software of [Ledoux et al., 2012]) to repair individual polygons before the application of SR. These measures can contribute to improving the automated level of the prototype. Another aspect that can be further improved is the selection of an appropriate tolerance value. The proper value of tolerance often varies and relies on the characteristics of the specific dataset. One potential approach is to conduct a preliminary analysis of the input dataset, such as examining adjacency relationships and computing distances between elements. Such analysis can provide a foundational understanding of the dataset and may offer suggestions for selecting an appropriate tolerance value. This needs to be further studied.

- **Compare with ISR on polygons**
  Traditional SR is typically based on a regular grid. It has the advantage of being simple to understand and implement. However it theoretically requires more space and time resources. The intention of developing an approach with a triangulation as a supporting data structure is to optimize the efficiency of traditional SR and offer more possibility and flexibility. The triangulation-based approach should be more efficient as only necessary elements are stored (some cells in the grid of traditional SR do not store any elements hence the grid is not fully utilized and there will be waste). However in order to preserve the attributes of polygons and retain the potential of handling overlap areas, additional time is required due to the utilization of BFS algorithm. Moreover, ISR can be adjusted to be applied on polygons by using the boundaries of polygons

as input line segments. Due to the limitation of time and computing resources, the comparison between the two has not been conducted. However, such a comparison would be highly beneficial and straightforward, allowing for a clearer understanding of the differences between the two approaches and providing valuable insights into the performance of the developed prototype.

Last but not least, if the input is assumed to be valid, the efficiency can be improved even more by altering the tagging stage. The constraints can be obtained in the reading process, i.e. when the input polygons are being read by GDAL. Such manner will reduce the run time to a large extent, however it can not handle the invalidities like self-intersections and overlaps.

Bibliography

(1985). Ieee standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20.

(1987). Ieee standard for radix-independent floating-point arithmetic. *ANSI/IEEE Std 854-1987*, pages 1–19.

Belussi, A., Migliorini, S., Negri, M., and Pelagatti, G. (2016). Snap rounding with restore: An algorithm for producing robust geometric datasets. *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, 2(1):1–36.

Bennett, J. (2010). *OpenStreetMap*. Packt Publishing Ltd.

Bentley, J. L. and Ottmann, T. A. (1979). Algorithms for reporting and counting geometric intersections. *IEEE Transactions on computers*, 28(09):643–647.

Boissonnat, J.-D., Devillers, O., Teillaud, M., and Yvinec, M. (2000). Triangulations in cgal. In *Proceedings of the sixteenth annual symposium on Computational geometry*, pages 11–18.

Bundy, A. and Wallen, L. (1984). Breadth-first search. *Catalogue of artificial intelligence tools*, pages 13–13.

Carter, C. (2002). Great circle distances. *SiRF White Paper*.

Chew, L. P. (1987). Constrained delaunay triangulations. In *Proceedings of the third annual symposium on Computational geometry*, pages 215–222.

Fabri, A. and Pion, S. (2009). Cgal: The computational geometry algorithms library. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 538–539.

Forsythe, G. and Moler, C. (1967). Computer solution of linear algebraic systems." prentice-hall, englewood cliffs, new jersey.

Fortune, S. (1998). Vertex-rounding a three-dimensional polyhedral subdivision. In *Proceedings of the fourteenth annual symposium on Computational geometry*, pages 116–125.

GDAL/OGR contributors (2022). *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation.

GeoFabrik (2023). Geofabrik: Download server for openstreetmap data. Web Based Download Application: `http://download.geofabrik.de/`. Last checked on April 2023.

GEOS contributors (2021). *GEOS coordinate transformation software library*. Open Source Geospatial Foundation.

Gold, C. M., Nantel, J., and Yang, W. (1996). Outside-in: an alternative approach to forest map digitizing. *International Journal of Geographical Information Science*, 10(3):291–310.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48.

Goodrich, M. T., Guibas, L. J., Hershberger, J., and Tanenbaum, P. J. (1997). Snap rounding line segments efficiently in two and three dimensions. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 284–293.

Greene, D. H. and Yao, F. F. (1986). Finite-resolution computational geometry. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 143–152. IEEE.

Guibas, L. J. and Marimont, D. H. (1995). Rounding arrangements dynamically. In *Proceedings of the eleventh annual symposium on Computational geometry*, pages 190–199.

Haklay, M. (2010). How good is volunteered geographical information? a comparative study of openstreetmap and ordnance survey datasets. *Environment and planning B: Planning and design*, 37(4):682–703.

Halperin, D. and Packer, E. (2002). Iterated snap rounding. *Computational Geometry*, 23(2):209–225.

Hershberger, J. (2011). Stable snap rounding. In *Proceedings of the twenty-seventh annual symposium on Computational geometry*, pages 197–206.

Hobby, J. D. (1999). Practical segment intersection with finite precision output. *Computational Geometry*, 13(4):199–214.

Kettner, L., Mehlhorn, K., Pion, S., Schirra, S., and Yap, C. (2008). Classroom examples of robustness problems in geometric computations. *Computational Geometry*, 40(1):61–78.

Kukulska, A., Salata, T., Cegielska, K., Szylar, M., et al. (2018). Methodology of evaluation and correction of geometric data topology in qgis software. *Acta Scientiarum Polonorum. Formatio Circumiectus*, 17(1):137–150.

Laurini, R. and Milleret-Raffort, F. (1994). Topological reorganization of inconsistent geographical databases: a step towards their certification. *Computers & Graphics*, 18(6):803–813.

Ledoux, H., Arroyo Ohori, K., and Meijers, M. (2012). Automatically repairing invalid polygons with a constrained triangulation. In *Proceedings of the AGILE 2012 International Conference, April 2012, Avignon, France, pp. 13-18*. Agile.

Lloyd, E. L. (1977). On triangulations of a set of points in the plane. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 228–240. IEEE.

Mulmuley, K. (1994). Computational geometry. an introduction through randomized algorithms. *(No Title)*.

Ohori, K. A., Ledoux, H., and Meijers, M. (2012). Validation and automatic repair of planar partitions using a constrained triangulation. *Photogrammetrie-Fernerkundung-Geoinformation*, 5(10):613–630.

o'Rourke, J. (1998). *Computational geometry in C*. Cambridge university press.

Packer, E. (2006). Iterated snap rounding with bounded drift. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 367–376.

QGIS Development Team (2009). *QGIS Geographic Information System*. Open Source Geospatial Foundation.

Raab, S. (1999). Controlled perturbation for arrangements of polyhedral surfaces with application to swept volumes. In *Proceedings of the fifteenth annual symposium on Computational geometry*, pages 163–172.

Schirra, S. (1998). Robustness and precision issues in geometric computation.

Sehra, S. S., Singh, J., and Rai, H. S. (2016). Analysing openstreetmap data for topological errors. *International Journal of Spatial, Temporal and Multimedia Information Systems*, 1(1):87–101.

Sehra, S. S., Singh, J., Rai, H. S., and Anand, S. S. (2020). Extending processing toolbox for assessing the logical consistency of openstreetmap data. *Transactions in GIS*, 24(1):44–71.

Shewchuk, J. R. (1996). Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In Lin, M. C. and Manocha, D., editors, *Applied Computational Geometry Towards Geometric Engineering*, pages 203–222, Berlin, Heidelberg. Springer Berlin Heidelberg.

Tjiharjadi, S. and Setiawan, E. (2016). Design and implementation of a path finding robot using flood fill algorithm. *International Journal of Mechanical Engineering and Robotics Research*, 5(3):180–185.

Ubeda, T. and Egenhofer, M. J. (1997). Topological error correcting in gis. In Scholl, M. and Voisard, A., editors, *Advances in Spatial Databases*, pages 281–297, Berlin, Heidelberg. Springer Berlin Heidelberg.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.